Area Paper:
A Framework for Hybrid Markov Logic

Tim Kopp

May 13, 2014
Abstract

Purely logical artificial intelligence systems are very good at exploiting strong relational structure in the domain of interest. However, they are brittle when we want to incorporate uncertainty into our models. Conversely, purely statistical methods are very good at dealing with uncertainty in models, but fail to efficiently reason about strong relational structures. As a result, statistical relational learning systems have become a prominent topic in recent artificial intelligence research. Statistical relational learning systems are systems that attempt to incorporate both the ability to deal with uncertainty and exploit strong relational structure when reasoning about the domain of interest.

Markov logic networks are a prominent statistical relational learning framework. A Markov logic networks is a set of weighted first-order logic formulas that describe a probability distribution across a finite domain. They have been used with great success for modeling a variety of systems. Recently, a system of hybrid Markov logic networks has been proposed for reasoning about continuous domains and non-i.i.d. data.

In this area paper we propose a framework for augmented hybrid Markov logic networks based on SAT modulo theory engines. We start with a discussion of SAT and SMT solvers, then give an in-depth description of the work that has been done on Markov logic networks. In the last section, we describe hybrid Markov logic networks and how to augment them to reason about decidable first-order theories. We propose some algorithms for inference on these augmented hybrid Markov logic networks.
# Contents

1  SAT and SMT  3  
1.1  Propositional & First-Order Logic  
1.1.1  Propositional Logic  3  
1.1.2  Normal Forms  4  
1.1.3  First-Order Logic  4  
1.2  The SAT Problem, SAT Solvers, and Related Material  6  
1.2.1  Propositional Satisfiability Problem  6  
1.2.2  Backtracking Search Solvers  7  
1.2.3  MaxSAT  9  
1.2.4  Sampling Satisfying Assignments  10  
1.3  The SMT Problem, SMT Solvers, and Related Material  11  
1.3.1  Sat Modulo Theories  11  
1.3.2  Eager SMT Solvers  12  
1.3.3  Theory Solvers  13  
1.3.4  Lazy SMT Solvers  15  
1.3.5  MaxSMT  16  

2  Markov Logic  19  
2.1  Conditional Random Fields  19  
2.1.1  Inference  20  
2.2  Markov Logic Defined  21  
2.2.1  Formal Definition  22  
2.2.2  Typical Assumptions  23  
2.3  Inference  23  
2.3.1  Most Probable Explanation Inference  23  
2.3.2  Conditional Probabilities  24  
2.4  Weight Learning  25  
2.4.1  Generative Weight Learning  25  
2.4.2  Discriminative Weight Learning  27  
2.5  Related Work  28  

3  Hybrid Markov Logic — a Proposed Framework  30  
3.1  Hybrid Markov Logic — State of the Art  30  
3.1.1  Definition  30  
3.1.2  MPE Inference in HMLNs  31
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.3</td>
<td>Conditional Inference in HMLNs</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>A New Formulation for Hybrid Markov Logic Networks</td>
<td>33</td>
</tr>
<tr>
<td>3.2.1</td>
<td>An SMT-like Formulation for Hybrid Markov Logic Networks</td>
<td>33</td>
</tr>
<tr>
<td>3.2.2</td>
<td>MPE Inference in AHMLNs</td>
<td>34</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Conditional Inference in AHMLNs</td>
<td>34</td>
</tr>
</tbody>
</table>
Chapter 1

SAT and SMT

In this chapter we discuss the Propositional Satisfiability and SAT Modulo Theories problems, as well as algorithms for these problems. We start with a section on propositional and first-order logic, follow with a section on the SAT problem, and finish with a discussion of the SMT problem.

1.1 Propositional & First-Order Logic

In this section we briefly introduce propositional and first-order logic. These concepts will be important for understanding subsequent sections. The concepts of propositional and first-order logic have been developed over millenia. We refer the reader to [17, 22] for a more complete overview of these logics and their development.

1.1.1 Propositional Logic

We define a Boolean variable, or propositional variable as a variable that can take on one of two values, true or false, often represented as 1 and 0 respectively.

In propositional logic, negation (¬) is an operation that changes the truth value of a variable: if \( x_1 \) is assigned true, then \( \neg x_1 \) is false. We refer to a variable or its negation as a literal.

Next we define a propositional connective, a \( \{0,1\} \)-valued functions that takes one or more Boolean variables, and produces either 1 or 0. The propositional connectives we will consider are logical and or conjunction (\( \land \)), logical or or disjunction (\( \lor \)), implication (\( \implies \)), and logical xor (\( \oplus \)). The definitions of these connectives, given as truth tables, are provided in Figure 1.1:

\[
\begin{array}{c|c|c|c|c}
 x & y & x \land y & x \lor y & x \implies y & x \oplus y \\
 0 & 0 & 0 & 0 & 1 & 0 \\
 0 & 1 & 0 & 1 & 1 & 1 \\
 1 & 0 & 0 & 1 & 0 & 1 \\
 1 & 1 & 1 & 1 & 0 & 0 \\
\end{array}
\]

Table 1.1: Truth Table for Propositional Connectives

A propositional formula consists of literals connected by logical connectives, e.g. \( (x_1 \land x_2) \implies \)
Given an assignment of the variables $\alpha : \{x_i\} \rightarrow \{0, 1\}$, a propositional formula can be evaluated to a single true or false value. We say an assignment satisfies a propositional formula if it causes the formula to evaluate to true.

A formula is a tautology if every assignment satisfies the formula. A formula is unsatisfiable if it has no satisfying assignment. A formula is satisfiable if there is at least one satisfying assignment.

### 1.1.2 Normal Forms

All propositional formulas have a particular structure. Certain structures have interesting properties, we call these structures normal forms. Here we introduce conjunctive normal form, disjunctive normal form, and horn clauses.

**Conjunctive Normal Form** A formula is in conjunctive normal form (CNF) if it is a conjunction of disjunctions of literals, e.g.

$$(x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_2) \land \neg x_3$$

If a formula is in this form, then one of the disjunctions is referred to as a clause. This form is special because in order to evaluate to true, all of the clauses need to evaluate to true. Furthermore, for a clause to evaluate to true, one or more of its literals must evaluate to true. Finally, every propositional logic formula can be converted to CNF.

**Disjunctive Normal Form** A formula is in disjunctive normal form (DNF) if it is a disjunction of conjunctions of literals, e.g.

$$(x_1 \land \neg x_2 \land x_3) \lor (\neg x_1 \land x_2) \lor \neg x_3$$

Intuitively, each conjunction, provided it does not contain the negated and non-negated form of the same literal is a satisfying assignment to the DNF formula. Furthermore, the list of such conjunctions in the formula is precisely the list of satisfying assignments. Every propositional formula can be converted to DNF.

**Horn Clauses** A clause is horn clause if it contains at most one non-negated literal. Intuitively, it is an implication whose antecedent is a conjunction of non-negated literals, and whose consequent is a single non-negated literal or is empty. A formula that is a conjunction of horn clauses (a special case of a CNF formula) can be reasoned about more effectively than general propositional logic formulas.

### 1.1.3 First-Order Logic

First-order logic (FOL), like propositional logic, is a logical calculus of truth values. It extends propositional logic with predicates, functions, constants, and quantifiers. First-order logic is more expressive than propositional logic, but also more difficult to reason about.

A first-order knowledge base (KB) is a set of first-order logic formulas. First-order logic formulas are constructed from logical connectives, quantifiers, and four types of symbols: constants, variables, functions, and predicates. A constant symbol is used to represent particular objects in the domain of interest. If we're reasoning about the spread of disease, they could represent people. A variable
symbol is a symbol that can take on the form of any object in the domain. A variable symbol needs an associated quantifier to be valid. The $\forall$ quantifier means that the formula holds for every object in the domain, whereas the $\exists$ quantifier means that the formula holds for at least one object in the domain. A function symbol represents a mapping from a tuple of domain objects to a single domain object. Constant symbols can be thought of as 0-ary function symbols. A predicate symbol represents a mapping from a tuple of domain objects to a Boolean value. Intuitively, predicates are used to talk about properties of objects, e.g. Smokes(Anna), or relations between objects, e.g. Friends(Anna,Bob). An interpretation of a FOL formula specifies which domain objects the constant symbols represent, as well as defining the relations represented by the function and predicate symbols.

A common extension of FOL is typing. In typed FOL, domain objects all have a single associated type, such as Person. As a result, all constants have a type, all predicates and functions take tuples of certain types, all functions return a certain type, and all variables are quantified only over a certain type.

In FOL, a term is any expression that represents an object in the domain. Thus constants, variables, and (even nested) function symbols are terms. An atomic formula, atom, or literal is a predicate symbol applied to a tuple of terms. A formula then, is a recursive construction of atoms connected with logical connectives (the same ones seen in propositional logic). Parenthesis may be used to enforce precedence. A positive literal is an atom. A negative literal is a literal that has been negated with $\neg$. In a first-order KB, the formulas are implicitly conjoined. As a result, the whole KB can be thought of as a single large formula. A ground term is a term containing no variables. Likewise, a ground predicate is a predicate containing no variables. A possible world is an interpretation: it assigns a truth value to each possible ground atom.

As in propositional logic, a formula is satisfiable if there is at least one possible world in which the formula is true. Likewise, the formula is a tautology if all of the possible worlds satisfy the formula, and is unsatisfiable if no possible worlds satisfy the formula. As in propositional logic, there are normal forms for first-order logic. In fact, the first-order logic normal forms are just the propositional logic normal forms with Boolean variables replaced with literals, and using only universal quantifiers at the outermost scope.

Inference in FOL  The typical inference task in FOL is to determine whether a knowledge base $\Delta$ entails a formula $\varphi$, written $\Delta \models \varphi$. We say $\Delta$ entails $\varphi$ iff $\varphi$ is true in all of the worlds in which $\Delta$ is true.

A common technique for inference is refutation. A formula $\varphi$ is entailed by $\Delta$ iff $\Delta \cup \neg \varphi$ is unsatisfiable. To test satisfiability in FOL, we use a technique called resolution [53]. The key intuition in resolution is that if we have two clauses $a_1 \lor \cdots \lor a_l \lor c \lor b_1 \lor \cdots \lor b_m$ and $d_1 \lor \cdots \lor d_n \lor \neg c \lor f_1 \lor \cdots \lor f_p$, then the clause $a_1 \lor \cdots \lor a_l \lor b_1 \lor \cdots \lor b_m \lor d_1 \lor \cdots \lor d_n \lor f_1 \lor \cdots \lor f_p$ is implied regardless of the assignment of $c$. We first convert $\Delta \cup \neg \varphi$ into CNF, and then apply this rule repeatedly. If the empty clause is ever derived, then the formula is unsatisfiable, meaning $\Delta \models \varphi$.

Inference is FOL is semidecidable; if $\Delta \models \varphi$, then resolution or another inference algorithm will eventually prove it. If not, however, the algorithm will run continually. However, certain subsets of FOL, such as formulas in Horn clause form, are decidable. Logical programming languages such as Prolog [36] take advantage of this face.
Theories

A FOL theory is a set of constant, predicate and function symbols along with a set of axioms, FOL formulas that constrain the definitions of the constants, predicates, and functions. For example, the theory of Presburger arithmetic defines the constant 0, unary function $S$ (for successor), binary function $Plus$, and a predicate for equality $Equals$. It’s axioms are:

1. $\forall x \neg Equals(S(x), 0)$
2. $\forall x \forall y Equals(S(x), S(y)) \implies Equals(x, y)$
3. $\forall x Equals(Plus(x, 0), x)$
4. $(\varphi(0) \land \forall x (P(x) \implies P(s(x)))) \implies \forall y P(y)$

In item 4, $\varphi(x)$ is any first-order formula with single free variable $x$. Presburger arithmetic was introduced by Mojżesz Presburger in 1929, but we refer you to [17] for an English explanation of the theory.

Presburger arithmetic is an example of a decidable theory. A decidable theory is a theory for which there exists an effective procedure (algorithm) for determining if an arbitrary set of formulas are consistent with the theory. We will deal with decidable theories again when we discuss the SAT Modulo Theories problem. However, first we will discuss the propositional satisfiability problem.

1.2 The SAT Problem, SAT Solvers, and Related Material

The Propositional Satisfiability (SAT) Problem is well-studied for both its theoretical and practical significance. The problem is NP-complete, making it a problem of great interest for theoreticians. Algorithms that solve this problem have many real-world applications, from planning to circuit verification. In this section we discuss the SAT problem and related problems, algorithms for those problems, and applications.

1.2.1 Propositional Satisfiability Problem

The Propositional Satisfiability (SAT) Problem was first introduced by Cook in 1971 [10] as the first example of an NP-complete problem. Here we formally introduce the SAT problem, describe what a SAT solver is, and discuss strategies used by the best existing solvers.

Definition 1.1. Propositional Satisfiability Problem.

Consider a propositional formula $\varphi$ with variables $\{x_1 \ldots x_n\}$. The Propositional Satisfiability Problem (SAT) is to determine whether or not there exists an assignment $\alpha$ of $\{x_1 \ldots x_n\}$ such that $\varphi$ is satisfied. A SAT solver is an algorithm that solves this problem.

Theoretical Considerations The SAT problem is NP-Complete, as proven by Cook and Levin in the early 1970s. A problem $A \in NP$ if it can be solved by nondeterministic machines in polynomial time. A problem $A$ is NP-Complete if $A \in NP$ and every $B \in NP$ can be converted into $A$ through a polynomial-time transformation of $B$’s inputs. This links SAT to the open P vs. NP problem because SAT is known to be in NP, but not known to be in P. If it can be proven that SAT (or any other NP problem) is not in P, that is, it has no polynomial-time algorithm, then the class NP is a proper superset of P. Conversely, if a polynomial-time algorithm is found for an NP-Complete problem such as SAT, then P and NP are the same set.
**Practical Considerations**  Algorithms for SAT are useful in many practical applications, including electronic design automation, planning, and formal equivalence checking [38]. This is because these problems are in NP and can therefore be converted into equivalent propositional formulas using polynomial-time transformations.

**SAT Competition**  The SAT community holds regular SAT competitions where different solvers compete to solve formulas from application, random, and crafted domains most efficiently [4].

**SAT Solvers** A SAT solver is an algorithm that solves the SAT problem or some restriction of it. Typically, SAT solvers are designed to solve propositional formulas that are given in CNF. Note that all propositional formulas can be transformed into an equivalent formula in CNF. In this section we will discuss the two main kinds of SAT solvers, backtrack-search solvers and local search solvers, and then discuss extensions to these algorithms and less common approaches.

### 1.2.2 Backtracking Search Solvers

In this section we discuss backtracking search solvers, a type of SAT solver under which most state-of-the-art solvers fall.

**The Backtracking Search Algorithm**  The Davis-Putnam-Logemann-Loveland algorithm (DPLL) [12], is the base upon which most modern solvers are built. The DPLL algorithm is a refinement of the resolution-based Davis-Putnam Procedure (DP) [13]. The DP algorithm works by recursively selecting a literal to assign, and checking that adding that literal to the partial assignment does not cause the formula to become unsatisfiable. If this occurs, the algorithm backtracks and assigns the opposite literal. If this then fails, then the algorithm returns unsatisfiable to the previous level of recursion. This algorithm is complete, meaning that if the SAT instance is satisfiable, it is guaranteed to find a satisfying assignment.

The DPLL algorithm refines this approach with two techniques. The first is called boolean constraint propagation (BCP), unit clause propagation, or simply unit propagation. This technique exploits the observation that unit clauses, clauses with precisely one literal, allow us to make trivial deductions. For example, in the formula \( x_1 \land (\neg x_1 \lor x_2) \land \ldots \), it is clear that if the formula is satisfiable, then \( x_1 \) is assigned true in every satisfying assignment. Furthermore, the same principle can be applied to a clause in which the partial assignment has assigned all but one of the literals and not satisfied the formula. For example, in the formula \( (x_1 \lor \neg x_2 \lor x_3) \land \ldots \) under the partial assignment \( \{x_2, \neg x_3\} \), it is clear that if formula is satisfiable under the partial assignment, then \( x_1 \) is assigned to true in all satisfying assignments derived from the partial assignment. The DPLL algorithm continually applies this rule after assigning a variable until no more unit clauses can be found.

The Boolean constraint propagation technique greatly increases the efficiency of the original DP algorithm. Researchers at Princeton University recognized that SAT solvers spent about 90% of their running time running the BCP procedure. They introduced an important optimization to BCP called two-literal watch in their Chaff SAT solver [44]. It works as follows, in each unsatisfied clause, choose two literals to watch. If neither literal is assigned false, the clause is not implied. If one or both of the watched literals is assigned false, then attempt to replace them with literals

---

not yet assigned false. If after this attempt, we have one watched literal, the clause is implied, and the implication is the assignment that makes the watched literal true. If we have zero watched literals, we have found a contradiction, and know that our partial assignment is not satisfying. If we have two watched literals, our clause is not implied. This optimization made Chaff very efficient, and two-literal watch schemes and variants are used in almost all modern DPLL-based solvers.

The second technique, called pure literal elimination exploits the fact that if a literal occurs in only one polarity, that is, it either always occurs negated or never occurs negated, then its variable can be trivially assigned in the polarity in which it occurs. Furthermore, if after applying a partial assignment to a formula, a literal occurs in only one polarity in clauses not-yet-satisfied, then that literal too can be trivially assigned to the polarity in which it occurs in the not-yet-satisfied clauses. This rule is applied after each variable assignment.

Clause Learning and Other Extensions to DPLL The standard DPLL algorithm has been extended with two additional techniques: clause learning and nonlinear backtracking. These techniques have greatly increased the efficiency of these SAT algorithms, making them the best SAT solvers for use with many (if not most) real-world applications.

Clause Learning is a process by which a backtracking SAT solver can learn constraints that are implied from the given formula as it solver. Consider a conflict, a case in which a backtracking solver finds that its partial assignment does not satisfy the formula. Clearly, the partial assignment at this time, which we can express as a conjunction of the assigned literals, is an implication for the unsatisfiability for the formula. Likewise, its negation is an implication for the satisfiability of the formula if it was originally satisfiable. Thus, we can create a new conflict clause by negating the partial assignment, and add it to our clause database. This technique was first pioneered in work done for explanation-based learning, e.g. [14].

Clause learning helps the efficiency of the solver in a few ways. The first is by adding clauses to the clause database. This directly increases the possibility of having unit clauses, which allows trivial inference and faster solving. Second, the new clauses provide information that can be used by the branch selection algorithm, the algorithm that decides which variable to assign next.

A third optimization enabled by clause learning is nonlinear backtracking. This technique is perhaps best illustrated by example. Suppose we have the following partial assignment assigned in the order \( \{x_1, \neg x_4, x_3\} \). Now suppose we reach a conflict at this point from the clause \( \neg x_1 \lor \neg x_3 \). We revert our last assignment and get \( \{x_1, \neg x_4, \neg x_3\} \). Now suppose for sake of argument that we reach another conflict from the clause \( \neg x_1 \lor x_3 \). In a traditional backtracking algorithm, we would revert our previous assignment and try \( \{x_1, x_4\} \). However, since neither of the two conflict clauses we reached involved \( x_4 \), we know that changing it from \( \neg x_4 \) to \( x_4 \) will still yield conflicts, so we can backtrack all the way to our first step and try \( \{\neg x_1\} \).

Clause learning also legitimizes the technique known as restarts [2]. Suppose a CNF formula is only satisfiable when variable \( x \) is assigned false, but this is not immediately obvious via deduction using standard techniques used by SAT solvers. The solver could assign \( x \) to true right away, meaning the solver would have to search \( 2^n - 1 \) possibilities (minus those eliminated through previously-discussed techniques). In other words, one poor choice early in solving can severely decrease performance. Restarts are designed to mitigate this issue. A solver that employs restarts starts solving and, if it does not find a solution after a certain time (where time could be measured in clock time, clauses learned, assignments made, etc.), then, it restarts solving with the empty assignment while retaining the clauses learned from the previous attempt at solving. With this extra information, it is hoped that the solver will make better decisions early in the solving

8
procedure, resulting in increased performance.

Although clause learning enables many optimizations and solving techniques, it comes with overhead. The number of clauses learned by the solver grows quickly, adding overhead to any searches of the clause database and other similar operations. Modern solvers employ clause database management systems, strategies for choosing which conflict clauses to keep and which to delete [14]. Note that since conflict clauses are simply implications for the satisfiability of the formula (that is, they do not contain any “new” information, they summarize information buried in layers of related clauses), there is no harm in deleting them. This allows the solver to keep the clause database to a reasonable size, and hopefully only keep the clauses which are most useful in making assignment decisions.

To summarize, backtracking solvers make up the most successfully solvers for use in formulas constructed from real-world problems. They are typically complete, and differ primarily in their strategies for choosing the next variable to assign (branching rule), their clause database management systems, and their strategies for determining when to restart.

Local Search Solvers

The other commonly-used SAT solving technique is local search. Local search algorithms are typically incomplete, or are not guaranteed to find a solution if one exists. Thus, they typically cannot be used to determine unsatisfiability. Local search solvers start with a base assignment, and repeatedly select a variable and reverse its polarity until the formula is satisfied. Local search solvers differ in their method for determining an initial assignment and the heuristic used to determine which variable to alter next.

GSAT is one such solver [55]. It works by randomly generating a starting assignment, and then continually flipping the assignment of the single variable that will cause the largest number of clauses to be satisfied. If after a predetermined number of flips a solution is not found, it starts over with a new random assignment. If after so many restarts a solution is not found, it gives up and returns unknown. GSAT performs very well on hard randomly-generated SAT instances. 3SAT instances, those formulas in which all clauses have precisely three literals, are considered hard if the ratio between the number of clauses and the number of variables is around 4.3. This figure was determined empirically, and is a point in which randomly generated formulas have about a 50% chance of being satisfiable [55].

WalkSAT [54] is an extension of GSAT. With certain probability p, WalkSAT will randomly choose a variable to flip. Otherwise, it will flip according to the strategy of GSAT. This algorithm was found to perform better than GSAT, and despite being introduced in the 1990s is still in widespread use today. It too performs well on randomly-generated formulas.

1.2.3 MaxSAT

Here we introduce the MaxSAT problem and variants, and describe both exact and approximation algorithms for the MaxSAT problem.

MaxSAT and Variants  The MaxSAT problem was first introduced in [33]. It can be thought of as the optimization version of SAT, and is defined below:

Definition 1.2. The MaxSAT problem is the problem of determining the largest number of clauses in a formula $\varphi$ that can be satisfied by a single assignment $\alpha$. 
A MaxSAT problem is **partial** if \( \varphi \) can be divided into two disjoint subsets, one of soft clauses, which need not be satisfied, and hard clauses, which must be satisfied. In partial MaxSAT, the solution is the assignment that satisfies the largest number of soft clauses while satisfying all of the hard clauses. Partial MaxSAT subsumes SAT — simply make the set of soft clauses empty. Partial MaxSAT also subsumes regular MaxSAT — simply make the set of hard clauses empty.

A MaxSAT problem is **weighted** if there is a real-valued weight associated with each clause. In weighted MaxSAT, the solution is the assignment that maximizes the sum of the weights of the satisfied clauses. Weighted MaxSAT subsumes regular MaxSAT — simply give all of the clauses equal weight.

A MaxSAT problem may be both partial and weighted. In weighted partial MaxSAT, the solution is the assignment which maximizes the sum of satisfied soft clauses while satisfying all of the hard clauses. Partial weighted MaxSAT subsumes, SAT, regular MaxSAT, partial MaxSAT, and Weighted MaxSAT.

Mathematically, the partial weighted MaxSAT problem is the problem of finding assignment \( \alpha^* \) such that \( \alpha^* \) satisfies all hard formulas and

\[
\alpha^* = \arg \max_{\alpha} \sum_i w_i I_i(\alpha)
\]

where \( i \) ranges over all of the soft clauses, \( w_i \) is the weight of the clauses, and \( I_i(\alpha) \) is an indicator function that is 1 if clause \( i \) is made true by \( \alpha \), and 0 otherwise.

**MaxWalkSAT** MaxWalkSAT is a version of WalkSAT that has been modified to solve the partial weighted MaxSAT problem [29]. Rather than flipping the variable that will satisfy the largest number of previously-unsatisfied clauses, it flips the variable that will maximize the sum of weights of newly-satisfied clauses.

Since it is a local search algorithm, MaxWalkSAT is not guaranteed to find the correct solution in any bounded amount of time (like WalkSAT). However, it can give an approximate solution, which improves given more iterations.

**Exact Algorithms** There are many exact algorithms for the MaxSAT family of problems, based on Branch-and-Bound or satisfiability testing. In practice, most applications in which MaxSAT solvers are used can tolerate an approximate solution more than they can tolerate the long running times of exact algorithms. As a result, these algorithms are not often used for real-world applications. In later chapters we will encounter the reduction of certain problems to MaxSAT. In all such instances, the use of MaxWalkSAT or a similar local-search algorithm is standard practice.

### 1.2.4 Sampling Satisfying Assignments

A propositional formula often has more than one satisfying assignment. In certain situations we want to sample the satisfying solutions uniformly. Suppose our propositional formula models how a digital circuit works. If we want to test our real-world circuit, it would take too long to test the exponential number of inputs. However, if we sampled satisfying solutions of the formula uniformly, we could generate a small number of test cases that we hope are a representative sample.
Wei et. al. showed that heuristic search strategies such as WalkSAT do not sample solutions in a uniform way, and present the SampleSAT algorithm, which samples solutions near-uniformly \[61\]. SampleSAT works by using a hybrid strategy of WalkSAT and simulated annealing \[32\].

With probability \(p\), the search performs a simulated annealing step. With probability \(1 - p\), the search performs a WalkSAT step. The authors proved that this strategy samples uniformly from the set of satisfying solutions, and found empirical evidence to support their claim.

XORSample

Gomes et. al. presented XORSample, a SAT-based technique for sampling satisfying solutions \[26\]. The system uses a SAT solver as a subroutine, and works by repeatedly modifying the formula to be sampled from, and running the solver on it.

As the name implies, XORSample works by adding a set of random XOR constraints to the formula. An XOR constraint is one of the form \(x_i \oplus x_j\), which states that precisely one of \(x_i\) and \(x_j\) must be true. By adding XOR constraints to the formula, the search space is significantly reduced (by half for each constraint added). When all of the constraints have been added, a SAT solver is repeatedly invoked, each time negating the solution of the previous call, until there are no solutions. The last solution that is output is returned as the sample. Alternatively, a model counter may be used.

The authors prove that this algorithm can sample arbitrarily-near to uniform based on the parameters that are used to choose the XOR constraints. Empirically, they find that XORSample samples much closer to uniformly than SampleSAT.

1.3 The SMT Problem, SMT Solvers, and Related Material

The SAT Modulo Theories Problems (SMT) are a family of well-studied constraint satisfaction problems. They can be thought of “first-order SAT” problems. We discuss the SMT problem, algorithms for those problems, and other related problems.

1.3.1 Sat Modulo Theories

Here we formally introduce the SAT Modulo Theories (SMT) family of problems. Although satisfiability testing for general FOL is undecidable, it is often the case that we want to test satisfiability for formulas that use a particular background theory. For example, if we are interested in doing integer arithmetic, we do not care to consider non-standard definitions of the \(<\) predicate in the formula \(x < y\). Intuitively, SMT is the problem of testing satisfiability for FOL formulas that are restricted to use background theories for which we have decision procedures. Since SMT problems and solvers vary depending on the background theory used, it is hard to pinpoint the “first” introduction of the SMT problem, but one early example using the theory of equality with uninterpreted functions is \[7\]. A very thorough treatment of SMT and related topics can be found in the SMT chapter of \[4\].

Definition 1.3. SAT Modulo Theories Problem.

Let \(\mathcal{T}\) be a set of first-order theories, and \(\Delta\) be a first-order knowledge base that use the theories in \(\mathcal{T}\). The SAT Modulo Theories (SMT) problem is the problem of determining if there exists an assignment \(\alpha\) that satisfies all of the formulas in \(\Delta\).
An algorithm that solves the SMT problem is called an SMT solver. There are two main approaches to SMT-solving. Eager SMT-solvers are those that encode the FOL theory in propositional logic, and pass the encoding to a SAT-solver. By contrast, Lazy SMT-solvers are those that deal with the SMT formula directly, typically through the combined use of a Boolean engine, such as a SAT solver, and a theory solver; a program for checking satisfiability of conjoined predicates in a FOL theory.

1.3.2 Eager SMT Solvers

An eager SMT solver works by taking an SMT knowledge base, encoding it to an equi-satisfiable propositional knowledge base, and testing satisfiability on it. The efficiency of the SMT solver depends on the propositional formula generated by the encoding. In general, the smaller and simpler the encoding is, the more efficiently it can be tested for satisfiability, but the more time it takes to perform the encoding.

Since eager SMT involves encoding the knowledge base directly into propositional logic, the techniques used in encoding depend directly on the theory $T$ in question. Eager SMT encodings have been demonstrated for the theories of

- Equality and Uninterpreted Functions
- Integer Linear Arithmetic
- Restricted Lambda Expressions, such as arrays
- Finite-Precision Bit-Vector Arithmetic
- Strings

We give an encoding from the theory of equality and uninterpreted functions to propositional logic as an example of an eager SMT encoding.

The first step of the encoding is to remove function symbols. Suppose we have three objects in the domain \{a, b, c\} and one unary function symbol $f$. This means that in the formula, we can see constraints with $f(a)$, $f(b)$, and $f(c)$. The Ackermann reduction is a technique for removing the function symbols. We replace every ground function term with a new variable, e.g. $f(a)$ is replaced by $A$. Then, we add the equality constraints so that for each pair of objects, if they are equal, then the functions of them are equal, e.g. $a = b \implies A = B$.

The second step is to encode the equality constraints in propositional logic. A naive encoding would be to add a variable $v_{a,b}$ for each constraint $a = b$, and then add the appropriate transitivity constraints $v_{a,b} \land v_{b,c} \implies v_{a,c}$.

An alternative encoding for the equality constraints is as follows. Suppose there are $n$ difference constants. We want to express each one as a binary number. In the worst case, none of them are equal, so we need $\log_2 n$ to express all of them. For each constraint $a = b$, we have a new variable $v_{a,b}$, and a constraint $v_{a,b} \iff \bigwedge_{i=0}^{\log_2 n} \neg (b_i^a \oplus b_i^b)$.

After building the encoding, we can use a SAT solver to test satisfiability. If we are interested in the model generated by the solver, we can determine which objects were equal by finding which $v_{a,b}$ were set to true.

There are many eager SMT solvers available, such as UCLID [35], STP [21], Spear [1], Boolector [6], and Beaver [28].
1.3.3 Theory Solvers

Let $T$ be a decidable FOL theory. A set of literals $\Phi$ is $T$-satisfiable iff there exists some model $M$ of $T$. Given a FOL theory $T$, a theory solver for $T$ is an algorithm that takes a set $\Phi$ of literals and returns whether or not $\Phi$ is $T$-satisfiable. A theory solver for $T$ is also known as a $T$-solver, or a decision procedure for $T$.

We give a short list of decidable theories commonly implemented by SMT solvers with short descriptions, then we describe an example theory solver in detail. After that, we describe how two solvers for different theories can be combined to create a solver for the union of two theories $T = T_1 \cup T_2$.

- Equality — the theory with only equality and uninterpreted functions.
- Linear Integer Arithmetic — the theory with the $\leq$ predicate and $+$, and $-$ functions on integer variables.
- Linear Real Arithmetic — the theory with the $\leq$ predicate and $+$, and $-$ functions on real variables.
- Difference Logic — a restriction of linear arithmetic where atoms take the form $x - y \leq c$.
- Fixed-size Arrays — provides the functions read and write on indexable array objects.
- Bit-vectors — similar to arrays, but includes functions for bit-wise operations.
- Tuples and Records — provides functions and predicates for reasoning about objects that are collections of other objects.

Equality-solver We present the theory solver given in [16, 47]. Let $T_\approx$ be the theory with equality and uninterpreted functions. Let $\Phi = \Gamma \cup \Delta$, where $\Gamma$ is the set of equalities, and $\Delta$ be the set of disequalities.

Let $\sim$ be the equivalence relation on $\Phi$ induced by $\Gamma$. Intuitively, if $\Gamma$ constrains two terms to be equal, then those two terms are in the equivalence relation. More formally, $t_1 \sim t_2$ iff $t_1 = t_2 \in \Gamma$ or $t_2 = t_1 \in \Gamma$. It follows from the definition that $t_1 \sim t_2$ iff $t_2 \sim t_1$. An equivalence class is a set of terms such that for every $t_1$ and $t_2$ in the set, $t_1 \sim t_2$. It follows from the definition that equivalence classes are disjoint.

The problem of solving this theory is the problem of determining which terms are in which equivalence class. To do this, we use standard disjoint set data structures, also known as union-find algorithms [20], so called because of the two operations used by the algorithm. The $\text{union}$ operation takes two elements, and places them in the same equivalence class by a set union operation. The $\text{find}$ operation returns an element representative of the equivalence class. The returned element will always be the same for a given class until another $\text{union}$ operation is performed.

Pseudocode for the solver is given in Algorithm 1. The algorithm works in two steps. It first performs all of the unions given by the constraints in $\Gamma$. It then uses the $\text{find}$ operation to check each of the disequalities in $\Delta$ to ensure that they are not in the same equivalence class.
Algorithm 1 $T_2$-solver

```plaintext
procedure $T_2$-solve($\Gamma, \Delta$)
    for $t_1 = t_2 \in \Gamma$ do
        union($t_1, t_2$)
    end for
    for $t_1 \neq t_2 \in \Delta$ do
        if find($t_1$) = find($t_2$) then
            return false
        end if
    end for
    return true
end procedure
```

Combining Solvers  Suppose we have two theories $T_1$ and $T_2$, and we have solvers for each of them. We may want to reason about a domain that uses predicates, functions, constants, and axioms from both domains, $\mathcal{T} = T_1 \cup T_2$. We would like to reuse the existing $T_1$ and $T_2$ solvers, rather than having to come up with an entirely new solver for $\mathcal{T}$. The Nelson-Oppen procedure [46] allows us to do this.

The Nelson-Oppen procedure can be used if the theories have no shared symbols other than equality, the theories are stably-infinite, and the formulas are quantifier-free. A theory $\mathcal{T}$ is stably-infinite if every $\mathcal{T}$-satisfiable quantifier-free formula is satisfiable in an infinite model.

Suppose $T_1$ and $T_2$ are theories for which we have solvers SAT$_1$ and SAT$_2$, respectively. To determine if a set of predicates $\phi$ is $T_1 \cup T_2$-satisfiable, we

1. Split $\phi$ into two disjoint subsets $\phi_1$ and $\phi_2$, where $\phi_1$ contains all of the predicates in $T_1$, and $\phi_2$ contains all of the predicates in $T_2$, respectively.
2. Let $S$ be the set of variables shared between $\phi_1$ and $\phi_2$.
3. For each arrangement $\Delta_i$ of $S$, $b_i = \text{Sat}_1(\phi_1 \cup \Delta_i) \land \text{Sat}_2(\phi_2 \cup \Delta_i)$
4. $\phi$ is $T_1 \cup T_2$-satisfiable iff there is some $i$ such that $b_i$ is true.

Theory Solvers for SMT  Theory solvers are able to determine satisfiability of a conjunction of literals. The SMT problem is more general, as the literals can have arbitrary logical structure. As we will see in later sections, theory solvers are still useful as a subroutine for SMT solvers. Here we list a few qualities that a theory solver may have that would make it more suitable for use in an efficient SMT-solver. This list was taken from [3].

- **Incrementality**: the ability to easily add new literals to $\phi$, or backtrack to a previous state of $\phi$. Typically this means that if $\phi$ is solved and then augmented with more literals to make $\phi'$, solving $\phi'$ is more efficient then if the solver had not previously solved $\phi$.
- **Layered/Lazy**: the ability to quickly detect simple inconsistencies quickly, possibly through the use of heuristics.
- ** Equality Propagating**: the ability to detect that two terms are equivalent, and exploit that knowledge to run more efficiently.
• **Model Generating**: the ability to output a satisfying model for satisfiable \( \phi \), rather than just reporting satisfiable/unsatisfiable.

• **Proof Generation**: The ability to output a proof of unsatisfiability for unsatisfiable \( \phi \). Typically the proof is output in a form usable by other theorem-proving systems.

• **Interpolant Generation**: The ability output Craig interpolants \[11\] for unsatisfiable formulas. Suppose \( \phi_1 \implies \phi_2 \) is unsatisfiable. An interpolant is a formula \( \alpha \) containing only symbols appearing in both \( \phi_1 \) and \( \phi_2 \) such that both \( \phi_1 \implies \alpha \) is unsatisfiable and \( \alpha \implies \phi_2 \) is unsatisfiable.

### 1.3.4 Lazy SMT Solvers

A *lazy* SMT solvers is one that deals with input formula \( \varphi \) directly, rather than compiling it to propositional logic. Most lazy SMT solvers use a variant of the \( \text{DPLL(T)} \) algorithm, which is itself a variant of the \( \text{DPLL(T)} \) algorithm used by some SAT solvers. In this section we give a description of \( \text{DPLL(T)} \) and discuss some optimizations to \( \text{DPLL(T)} \).

#### The \( \text{DPLL(T)} \) Algorithm

The \( \text{DPLL(T)} \) algorithm is an algorithm for SMT that uses a theory solver for \( T \) first introduced by Nieuwenhuis et. al. in \[50\]. It can be thought of as a straightforward extension to the \( \text{DPLL} \) algorithm used by many SAT-solvers. A \( \text{DPLL} \)-like procedure is run until satisfying assignment is found, and then a theory-solver is run on the answer to make sure the constraints of the theory are met.

More formally, let \( T \) be a first-order theory with theory solver \( \text{SAT}_T \), and \( \varphi \) be a \( T \)-formula in CNF for which we wish to test satisfiability in \( T \). We first create the Boolean abstraction \( \varphi^B \) of \( \varphi \), a propositional formula that is created by replacing each \( T \)-literal in \( \varphi \) with new Boolean variable.

We then run a modified \( \text{DPLL} \) procedure on \( \varphi^B \). When a satisfying solution is found, the assignment is passed to the \( T \)-solver. If the \( T \)-solver returns true, then the assignment \( T \)-satisfies \( \varphi \). If not, the search backtracks as if it did not satisfy \( \varphi^B \). The \( \text{DPLL} \) procedure has to be modified in the following ways to work properly on Boolean abstractions of SMT formulas:

- When we reach a state where the \( \varphi^B \) is satisfiable, but \( \text{SAT}_T \) returns false, we add the negation of the assignment to the formula as a conflict clause.
- The search does not apply the pure literal rule. The pure literal rule works because propositional literals are independent of one another. However, first-order literals may depend on one another.

Nieuwenhuis proves the \( \text{DPLL(T)} \) procedure to be sound and complete for SMT \[50\].

#### Optimizations to \( \text{DPLL(T)} \)

Here we give a few optimizations to the \( \text{DPLL(T)} \) algorithm as presented above to increase its efficiency. The techniques we discuss are the minimization of learned clauses, early conflict detection, and theory propagation.

As presented above, a learned clause is the negation of an entire assignment, meaning it is a clause with \( n \) literals, where \( n \) is the number of variables in the formula. It is often the case that the formula will be made unsatisfiable by only a subset of the variables in the assignment. *Learned clause minimization* is the process of taking learned clauses, and reducing their size. Doing so has two benefits. The first is storage space. By storing smaller clauses, the system will have lower
memory overhead. The second and far more significant benefit is that multiple possible assignments will have the same minimal conflict clause. Thus, by storing the minimal clause, we will remove all of the assignments who share the minimal conflict clause from the search space. One general technique for minimizing the conflict clause is to traverse the proof tree generated by the inconsistency. There also exists certain theory-specific procedures that can be used for clause minimization.

As presented above, the $T$-solver is only invoked when a complete assignment is made at a leaf in the search tree. An alternative approach would be to invoke the $T$-solver higher in the search tree to detect conflicts sooner. One can imagine invoking the $T$-solver as frequently as after each decision literal, or less frequently. Those theories with incremental $T$-solvers as discussed above, can greatly benefit from checking the $T$-satisfiability of the partial assignment frequently, since on each invocation it need only push the new literals and invoke the solver. SMT solvers that implement this strategy perform better than those that do not.

Unit propagation is the process of deducing literals from the given formula and partial assignment. Given a partial assignment and a $T$-formula, certain theory solvers can return a list of literals that are implied by the partial assignment in the $T$-formula. By running such a $T$-solver after each decision literal, we get a set of literals we can immediately assign. This technique works very well with the early conflict detection technique above. If no conflict is detected, then we get some new literals to assign in addition to the assurance that this branch in the tree is worth searching.

1.3.5 MaxSMT

The MaxSMT problem is analogous to the MaxSAT problem for SMT. Given a set of $T$-formulas, determine the largest number of them that can be satisfied by a single assignment. Like MaxSAT, there is a partial version, in which some formulas can be hard, a weighted version, in which different formulas are given different weights, and a partial weighted version, in which formulas are given real weights or weight infinity. The problem was first introduced in [49].

Until recently, MaxSMT has received relatively little attention in the literature. We present a reduction from MaxSMT to SMT given by [49] as a first means of solving the MaxSMT problem, and then present a direct algorithm that was discovered very recently.

Reducing (weighted, partial) MaxSMT to SMT Suppose we have an input $\Delta$ with a set of pairs $(\varphi_i, w_i)$, where $\varphi_i$ are $T$-formulas, and $w_i$ are real-valued weights or infinity. Intuitively, we use an SMT solver to iteratively ask “is there an assignment whose weight is at least $x$”, for different values of $x$, until we find the highest one.

In order to as the SMT solver “is there an assignment of at least weight $x$”, we give it $\Delta'$, a modified version of $\Delta$. $\Delta'$ starts empty. For each pair $(\varphi_i, \infty)$, we add $\varphi_i$ to $\Delta$. For every other pair, we add the following two constraints:

$$\varphi_i \implies x_i = w_i$$
$$\neg \varphi_i \implies x_i = 0$$

Finally, we add the constraint

$$x_1 + x_2 + \cdots + x_k \geq x$$

Obviously this technique requires the use of the theory of real linear arithmetic. However, since such a theory-solver exists, and since we showed how to use the union of two theories, this is no problem. This way of building $\Delta'$ makes it so that the satisfied formulas have corresponding $x_i$s
that are non-zero, and thus the last constraint will only be satisfied if there is an assignment to the variables that makes the total weight meet or exceed \( x \).

By changing the value of \( x \) on each query to the SMT solver, we can find the greatest value of \( x \) such that there is an assignment with that weight, completing the reduction to SMT. One could naïvely try all of the values between 1 and one plus the maximum weight, but it is more efficient to use a binary search, where the upper bound is the weight of the assignment if all of the formulas are satisfied (even if this is not possible). This involves making \( n \log n \) calls to the SMT solver, where \( n \) is the sum of the weights of the formulas.

**Lemma-lifting for MaxSMT** Cimatti et. al. recently introduced a lemma-lifting algorithm for MaxSMT [8]. The authors describe their algorithm as a “modular” approach, since it uses an SMT solver and a MaxSAT solver as black-boxes. The general idea is to alternatingly call the MaxSAT and SMT solvers, feeding information from the previous call into the next call.

As we introduced above, a \( T \)-conflict clause is a minimized clause returned by theory solver when a conjunction of \( T \)-literals is found to be unsatisfiable. Also introduced above, a \( T \)-deduction is a set of literals deduced through theory propagation. We use the term \( T \)-lemma to refer to either a \( T \)-conflict clause or a \( T \)-deduction. The lemma-lifting algorithm works by alternatingly 1) getting an assignment of literals that maximizes the weight of the Boolean abstraction (as introduced above) and 2) checking to see if such an assignment is \( T \)-consistent.

The SMT solver takes an assignment from the MaxSAT solver, and can be thought of as an enumerator of sets of \( T \)-lemmas, where the output of each successive iteration is a superset of the previous iteration’s output. The MaxSAT solver takes a set of \( T \)-lemmas from the SMT solver, and uses them to find the highest-weighted assignment that also satisfies the \( T \)-lemmas. It can therefore be thought of as an enumerator of variable assignments of decreasing weight. Algorithm 2 gives pseudocode for the lemma-lifting algorithm described in [8]. \( \varphi^T \ast \) refers to a set of \( T \)-formulas, and \( \psi^T \ast \) refers to the Boolean abstraction of that set of formulas. \( \varphi^h \ast \) refers to a set of hard clauses, and \( \varphi^s \ast \) refers to a set of soft clauses. The procedure \( \mathcal{T}2B \) produces the Boolean abstraction of a set of \( T \)-formulas, and likewise \( \mathcal{B}2T \) performs the inverse operation, producing a set of \( T \)-formulas from their Boolean abstraction. In the pseudocode, \( \theta^T \ast \) refers to a set of \( T \)-lemma clauses, and \( \psi^T \ast \) refers to an assignment of literals.

**Algorithm 2** Lemma-lifting for MaxSMT

\[
\begin{align*}
\text{procedure } & \text{MaxSMT}(\varphi^T \ast, \varphi^T \ast) \\
& \varphi^h \ast, \varphi^s \ast \leftarrow \mathcal{T}2B(\varphi^T \ast, \varphi^T \ast) \\
& \theta^T \ast \leftarrow \emptyset \\
& \psi^T \ast \leftarrow \varphi^T \ast \\
& \text{while SMT.solve}(\varphi^h \ast \cup \psi^T \ast \cup \theta^T \ast) = \text{UNSAT} \text{ do} \\
& \quad \theta^T \ast \leftarrow \theta^T \ast \cup \text{SMT.GetTLemmas}() \\
& \quad \theta^B \ast \leftarrow \mathcal{T}2B(\theta^T \ast) \\
& \quad \psi^B \ast \leftarrow \text{MaxSAT}(\varphi^h \ast \cup \theta^B \ast, \varphi^s \ast) \\
& \quad \psi^T \ast \leftarrow \mathcal{B}2T(\psi^B \ast) \\
& \text{end while} \\
& \text{return } \psi^T \ast \\
\text{end procedure}
\end{align*}
\]
Since the lemma-lifting algorithm uses an SMT solver and a MaxSAT solver as black boxes, some properties of the algorithm depend on which solvers are used. For example, it is common to use an approximation algorithm for MaxSAT. If such an algorithm is used, then the lemma-lifting algorithm will also produce an approximate solution. However, if an exact solver is used, an exact solution will be produced. Similarly, the theories supported by the lemma-lifting algorithm are precisely the theories supported by the SMT solver used.

The lemma-lifting algorithm can be made more efficient if the MaxSAT and SMT solvers have certain properties. If the MaxSAT solver is incremental, meaning that if after running on formula $\varphi_h \cup \varphi_s$, it can use information from that run to solve $\varphi_h \cup \varphi_s \cup \varphi'_h$, then the efficiency of the procedure is improved. Each call to the MaxSAT solver in the lemma-lifting algorithm is on the same set of soft clauses and a superset of the previous run’s hard clauses.

If the SMT solver is incremental, than we can make a less straightforward optimization. There is no set-theoretic relationship between the sets of satisfied clauses from iteration to iteration. However, we can augment each $T$-clause with a selection variable. In doing so, the incremental SMT solver will remember all of the soft clauses, and on each iteration we can “pop” the previous set of selection variables, and “push” the new set, operations almost universally supported by incremental SMT-solvers. In doing so, the SMT decision procedure runs more efficiently.

The authors prove the correctness of the lemma-lifting algorithm in [8]. They test it empirically against other MaxSMT solvers, and found that it typically does substantially better than other techniques.
Chapter 2

Markov Logic

In this chapter we define Markov Logic, give a number of algorithms for learning and inference within Markov Logic, discuss state-of-the-art Markov Logic systems, and give a number of real-world applications that use Markov Logic. We end with brief descriptions of other statistical-relational learning systems.

2.1 Conditional Random Fields

A Markov logic network can be thought of as a template for the creation of Markov networks, also known as conditional random fields. In this section we define conditional random fields, and briefly discuss associated algorithms for inference and learning. For a more in-depth discussion on conditional random fields, we refer the reader to [59].

Graphical models are used to model the joint probability distribution of a system \( p(y, x) \), where we wish to predict the values of the variables in \( y \) given observed values for the variables in \( x \). However, if the model uses a rich set of features, then the distribution \( p(x) \) can include many complex dependencies. Modeling these dependencies often makes the model intractable.

To deal with this problem Lafferty et al. introduced conditional random fields (CRFs) in 2001 [34]. A conditional random field, or undirected graphical model, or Markov network, is a graphical model in which the conditional distribution \( p(y|x) \) is modeled, rather than the joint distribution. By modeling the system in this manner, the (potentially complex) dependencies between the input variables \( x \) do not need to be represented explicitly, which means that richer feature sets can be used without causing the model to be intractable. We give a detailed description of conditional random fields.

Let \( V = X \cup Y \) be the set of random variables in the model, where \( X \) is the set of observed variables, \( Y \) is the set of variables to be predicted, and \( x \) and \( y \) are assignments thereof. For now we assume all of the random variables to be binary. The goal is to model the distribution over all of the variables as a product of local functions, each depending on a small subset of variables \( a \subset V \).

Theoretically, this can be exponential in the number of variables, but we use a smaller number of subsets for real-world models.

We introduce a function \( \Psi_a : V^{[a]} \to \mathbb{R}^+ \), called a factor, for each such subset, and call the set
of factors $F$. The conditional distribution is then

$$p(y|x) = \frac{1}{Z(x)} \prod_{a \in A} \Psi_a(x_a, y_a)$$

where $Z(x)$ is a constant normalization factor, known as the *partition function* and given by:

$$Z(x) = \sum_y \prod_{a \in A} \Psi_a(x_a, y_a)$$

For our purposes, we will assume that all of the factors are of the form

$$\Psi_a(x_a, y_a) = \exp \left\{ \sum_k w_{a,k} f_{a,k}(x_a, y_a) \right\}$$ (2.1)

This ensures that the family of distributions parameterized by $w$ is an exponential family.

Until this point we have used the terms *conditional random field* and *Markov network* interchangeably, however, they are subtly different. While a CRF, as the name implies, models the conditional distribution of a set of random variables on observed variables, a Markov network models the marginal distribution. However, the distribution is still modeled as a product of factors, so the equation is only subtly different:

$$p(x) = \frac{1}{Z} \prod_{a \in A} \Psi_a(x_a)$$ (2.2)

We now make no distinction between observed and query variables, and as a result the partition function is no longer a function of the observed variables. For the purpose of Markov logic networks, whether the model is a CRF or a Markov network depends on whether we are working in a discriminative or generative setting, and the difference will be clear from the context.

**Graphical Depiction of CRFs** It is often useful to think of a CRF as a graph (hence the name “graphical models”). A CRF can be thought of as a graph where there is a node for each variable in the CRF, and each $\Psi_a$ specifies a clique. That is, an edge between two nodes $u$ and $v$ if there is some $\Psi_a$ such that $u \in a$ and $v \in a$.

A CRF can be converted to a bipartite graph known as a *factor graph*. In addition to nodes for the variables, a factor graph contains a second kind of node for factors. One factor node is created for each clique (that is, each $\Psi_a$), and there is an edge from a variable node to a factor node iff the variable is in the clique specified by the factor.

**2.1.1 Inference**

There are two inference problems for CRFs. Marginal inference in the problem of finding or approximating the probability distribution $P(y|x)$. Maximum A Priori (MAP) inference is the problem of finding the most likely state of the random variables. As we said before, we are interested in CRFs because Markov logic networks are a template for their creation.
Belief Propagation  Belief propagation is a message passing algorithm for inference on graphical models that finds the conditional distribution of the model, the probability of the unobserved nodes given the observed nodes \[62\]. For a tree-structured CRF, belief propagation is an exact inference algorithm. For CRFs with loops, belief propagation produces approximate results and is not guaranteed to converge. However, belief propagation can still be used to efficiently obtain good estimates for loopy networks. Belief propagation is only briefly mentioned here, since it is not typically used for inference in Markov logic networks.

Markov-Chain Monte-Carlo Methods  The problem of marginal inference for CRFs is \#P-complete. To approximate the marginal distribution, a common technique is the use of Markov-Chain Monte-Carlo methods.

Markov-chain Monte-Carlo (MCMC) methods are algorithms for drawing samples from a particular distribution \(\pi\). In the context of Markov networks, we use MCMC methods to approximate the conditional distribution of the network \(p(y|x)\) by sampling from the distribution defined by the network, and reporting the proportion of samples in which \(y\) occurs.

We present the Metropolis-Hastings algorithm \[39\], a standard MCMC algorithm. We construct a Markov chain, a sequence of samples such that a particular sample depends only on the previous sample, i.e \(P(X_{t+1}|X_0,\ldots,X_t) = P(X_{t+1}|X_t)\). We must define \(P(\cdot|\cdot)\), known as the transition kernel of the chain. To obtain sample \(X_{t+1}\), we choose sample from the transition kernel, which can be done using a source of (pseudo) uniformly-random numbers.

After a certain number of iterations \(m\), known as the burn-in period, the samples will no longer depend on the initial state \(X_0\), and instead be dependent samples from the steady-state distribution \(\pi\). The Metropolis-Hastings algorithm gives us a means for ensuring that \(\pi\) corresponds to the distribution we wish to sample from.

In the Metropolis-Hastings algorithm, we sample a point \(Y\) from our transition kernel, \(q(\cdot|X_t)\). \(X_{t+1}\) is defined as follows:

\[
X_{t+1} = \begin{cases} 
Y & : \alpha(X_t,Y) \\
X_t & : 1 - \alpha(X_t,Y)
\end{cases}
\]

where

\[
\alpha(X,Y) = \min \left( 1, \frac{\pi(Y)q(X|Y)}{\pi(X)q(Y|X)} \right)
\]

In other words, we accept each sample with a probability that is related to \(\pi\). This ensures that after the burn-in period, all samples will be from \(\pi\). This approach works independently of \(q(\cdot|X_t)\).

MAP Inference  The MAP estimate of a CRF cannot be computed directly in closed form. However, the log-likelihood is a concave function of the weights, so the MAP estimate can be found using standard optimization techniques. The most common techniques are gradient-based or quasi-Newton optimization methods \[51\].

2.2 Markov Logic Defined

In this section we give a formal definition of Markov logic networks, describe frequently-used notational shortcuts, and give assumptions we typically make to ensure that everything is well-defined.
and finite. This section draws heavily from [15].

2.2.1 Formal Definition

**Definition 2.1.** A Markov logic network \( L \) is a set of pairs \((\varphi_i, w_i)\) where \( \varphi_i \) is a first-order logic formula and \( w_i \in \mathbb{R} \cup \{\infty\} \). It can be thought of as a template for the creation of Markov networks. Given a set of constants \( C = \{c_1, \ldots, c_{|C|}\} \), the Markov network \( M_{L,C} \) is defined as follows:

1. \( M_{L,C} \) contains a single binary node for each possible grouping of each predicate in \( L \). The value of the node is 1 iff the ground predicate is true.

2. \( M_{L,C} \) contains one binary feature for each possible grounding of each \( \varphi_i \in L \). The value of the feature is 1 iff the ground formula is true. The weight of the feature is \( w_i \).

**Constants** Since \( \varphi_i \in L \) are first-order logic formulas, they may contain constants. If this is the case, then it is necessary that any constant \( c \) appearing in any \( \varphi_i \) must also be in \( C \). It is common to refer to \( M_{L,C} \) as a ground Markov logic network. It should be clear that given a different set of constants \( C \), \( L \) will produce a different ground Markov logic network, hence its description as a template for creating Markov networks.

**Quantification** Since \( \varphi_i \in L \) are first-order logic formulas, they may contain quantifiers. For brevity, we always assume that free variables are universally quantified at the outermost scope.

It is worth describing item 2 of definition 2.1 in plain terms. If we consider the pair \((5.0, \forall x_1 P(x_1))\), and the set of constants \( C = \{c_1, c_2\} \), then the ground MLN contains the pairs \((5.0, P(c_1)), \) and \((5.0, P(c_2))\). In other words the weight attached to a universally quantified formula is the weight assigned to each grounding, not the weight assigned in total to all of the groundings. Soft universal quantification is “for each”, not “for all.”

**Possible Worlds** In this context, a possible world is a first-order logic interpretation: definitions of the functions and predicates that map all possible groundings to domain objects and truth values, respectively, as well as a mapping from constant symbols to domain objects. The score or weight of a possible world is defined as the sum of the non-infinite weights of ground formulas that are true in the world if there is no formula with infinite weight that is false in the world, and \(-\infty\) otherwise.

**Probability of Possible Worlds** The probability of a possible world can be thought of as the score of that world divided by the sum of the scores of all of the possible worlds. For a more formal definition, we think in terms of Markov networks.

Markov Logic networks are described in terms of factors of exponential families. By Equations 2.1 and 2.2 the probability distribution over the possible worlds \( x \) of the ground Markov network is

\[
P(X = x) = \frac{1}{Z} \prod_i \phi_i(x_{\{i\}})^{n_i(x)} = \frac{1}{Z} \exp \left( \sum_i w_i n_i(x) \right)
\]

where \( n_i(x) \) is the number of groundings of \( \varphi_i \) that are true in world \( x \), \( x_{\{i\}} \) is the set of truth values of the predicates in \( \varphi_i \), and \( \phi_i(x_{\{i\}}) = e^{w_i} \).  

22
2.2.2 Typical Assumptions

The definition given in the previous section can result in MLNs with an infinite number of possible worlds, or with a poorly-defined probability function. We often make the following assumptions, though if our domain is finite it is possible to eliminate them.

**Assumption 2.1. Unique names.** Different constants refer to different objects.

Consider what happens if we allow two constants to refer to the same domain object. When grounding the first-order MLN, each combination of constant symbols in the predicates is considered. Therefore, the same domain object is considered at least twice as often as it should be. Thus, when computing the partition function, we double-count worlds that have predicates with either of the constants, giving an incorrect probability. Implicit in this assumption is the fact that our domain must be finite.

**Assumption 2.2. Domain Closure.** There are no objects in the domain that cannot be represented with a constant symbol in $C$.

Consider what happens if we allow objects that we cannot represent. When grounding the first-order MLN, those objects are not included in the ground MLN, and therefore are not considered when calculating the partition function, artificially inflating the probabilities of the possible worlds that can be represented.

**Assumption 2.3. Known functions.** For any grounding of a function symbol in $L$, the value of that function with those arguments is known.

By Assumption 2.2, this means that not only is the value of a function with particular arguments known, but it is represented by an element of $C$.

This assumption is made for purposes of efficiency rather than correctness. When grounding the MLN, we can replace a ground function with its definition, e.g. $f(c_1, c_2, c_3)$ with $c_4$. If we did not make this assumption, we would have to encode all possible interpretations of the function in propositional logic, greatly increasing the size of the network.

2.3 Inference

In this section we discuss algorithms for MPE and marginal inference.

2.3.1 Most Probable Explanation Inference

Given a ground MLN, the *most probable explanation* (MPE) is the possible world that is most probable. Equivalently, it is the possible world with the highest score as defined in Section 2.2.1. It is almost always the case that we wish to find the most probable world $y$ given some evidence $x$, where $x$ is a set of non-conflicting literals. Formally,

$$\arg\max_y P(y|x) = \arg\max_y \frac{1}{Z(x)} \exp\left(\sum_i w_in_i(x, y)\right)$$

$$= \arg\max_y \sum_i w_in_i(x, y)$$
We get the first equality from Equations 2.2 and 2.1, and then the second because $Z(x)$ is constant and exponentiation is monotonically increasing.

We see that this equality is analogous to the formal definition of the MaxSAT problem given in Equation 1.1. Indeed, the problem of MPE inference in Markov logic straightforwardly reduces to the MaxSAT problem: the reduction is simply to ground the first-order MLN into a weighted propositional formula. Although the problem is NP-hard in general, there are a number of effective algorithms for exact and approximate MaxSAT. These are outlined in Section 1.2.3.

### 2.3.2 Conditional Probabilities

The problem of computing the probability that a FOL formula $\varphi_1$ holds given that formula $\varphi_2$ is readily answered by MLNs. We refer to $\varphi_2$ as the *evidence* formula or formulas, and $\varphi_1$ as the *query* formula. This problem is often referred to as the problem of *marginal* inference in MLNs. While this is technically incorrect unless $\varphi_2$ is a tautology, we will use it interchangeably with “computing the conditional probability.” We model it as the probability of $\varphi_1$ given both $\varphi_2$ and the ground MLN $M_{L,C}$:

$$P(\varphi_1|\varphi_2, M_{L,C}) = \frac{P(\varphi_1 \land \varphi_2|L_{L,C})}{P(\varphi_2|L_{L,C})} = \frac{\sum_{x:\varphi_1 \land \varphi_2} P(X = x|M_{L,C})}{\sum_{x:\varphi_2} P(X = x|M_{L,C})}$$

Computing this probability directly is intractable, but many approximation algorithms for marginal inference have been developed, some of which will be explained here. All of the approximations use the same general technique: sample from the distribution defined by the ground MLN given that $\varphi_2$ holds, and report the proportion of samples in which $\varphi_1$ holds as the conditional probability.

**Markov-Chain Monte Carlo Methods** Markov-chain Monte Carlo (MCMC) methods (Section 2.1.1) can be used to approximate $P(\varphi_1|\varphi_2, L, C)$ by rejecting all proposed states in which $\varphi_2$ does not hold. These techniques are typically too slow for general queries, however, we can increase the efficiency of MCMC inference by restricting the query formulas to be conjunctions of ground literals. By applying this restriction, we can perform MCMC inference while grounding only a subset of the MLN.

The ground network is constructed as follows: We check if all of the atoms the query directly depends on are grounded. If so, we stop. If not, we ground said atoms, and repeat. Once the network is constructed, standard MCMC methods, e.g. Gibbs sampling or Metropolis-Hastings can be used. While in the worst case this will provide no savings (the entire network is constructed), in practice time and memory requirements for inference are greatly reduced.

One drawback of the MCMC approach for MLNs is that the techniques break down for deterministic ($w_i = \infty$) and near-deterministic ($w_i \gg w_*$) constraints. Next we will see how to deal with such constraints.

**MC-SAT** In order for MCMC methods to work properly, the Markov chain must be *ergodic*, meaning that for all states $X_i$, the system must be able to reach state $X_j$ in a finite number of steps with non-zero probability. Deterministic constraints split the ground Markov network into two or more partitions that cannot be reached from one another, rendering MCMC methods ineffective.
Likewise, networks that have one or more constraints that have weight significantly greater than other constraints are split into two or more partitions that can only be reached from one another through very low-probability paths, making the burn-in time too long to be practical for real-world use.

MC-SAT [52] was created to address this problem. It is a slice-sampling MCMC algorithm. The general idea is that a SAT sampler is used to sample one of the disjoint subsets of the search space, and then MCMC methods are used to sample from within that space.

Recall that each clause \( c_k \) has a potential function \( \phi_k(x) = e^{w_k f_k(x)} \), whose value is \( e^{w_k} \) when \( c_k \) is satisfied, and 1 otherwise. For every ground clause in the Markov network \( c_k \) we introduce an auxiliary variable \( u_k \). Intuitively, on each iteration we sample values for all the \( u_k \), and a clause \( c_k \) is forced to be satisfied in the next sample if \( u_k \geq 1 \). We then sample an assignment that satisfies the forced clauses.

More formally, in iteration \( i \), if \( c_k \) is not satisfied by the assignment \( x^{(i-1)} \), then \( u_k \) is given a value drawn uniformly from \([0, 1]\). If \( c_k \) is satisfied by assignment \( x^{(i-1)} \), then \( u_k \) is given a value drawn uniformly from \([0, e^{w_k}]\), so that \( u_k > 1 \) with probability \( 1 - e^{-w_k} \). For each \( u_k > 1 \), \( c_k \) is added to set \( M \). Sample \( x^{(i)} \) is then obtained by sampling a satisfying assignment (see Section 1.2.4) of \( M \). Sample \( x^{(i)} \) is generated by using a SAT solver on the conjunction of the hard clauses in the network.

MC-SAT greatly outperforms traditional MCMC methods for Markov logic networks. Implementations of MC-SAT can be found in state-of-the-art Markov logic frameworks, such as Alchemy\(^1\) and Tuffy\(^2\).

### 2.4 Weight Learning

MLN weights can be learned generatively, meaning that the entire probability distribution for each variables is defined. By contrast, MLN weights can also be learned discriminatively, meaning that the probability distribution is defined only for query variables conditioned on observed variables. In this section we discuss both generative and discriminative weight learning algorithms.

#### 2.4.1 Generative Weight Learning

We learn weights generatively by maximizing the likelihood of data. The database can be thought of as a relational database that contains one or more possible worlds. We invoke Assumptions 2.1 and 2.2 so that the set of constants that appear in the database are assumed to be the entire set of constants. We next make the closed world assumption, meaning that all literals not named as true in the database are assumed to be false. Thus, a database can be thought of as a set of examples, where each example is a vector of ground atoms \( x = (x_1, \ldots, x_n) \) assigned a true or false value. Since each formula has only one weight and multiple groundings, weight learning can be performed effectively on even a single example.

With a database, MLN weights can be learned using standard methods. Let \( n_i(x) \) be the number of true groundings of formula \( i \) in the database. By Equation 2.3, the derivative of the log-likelihood

\[ \frac{d}{d\theta} \log p(x; \theta) = \sum_{i=1}^{m} \sum_{x_i \in \text{true}} n_i(x) \frac{\partial \phi_i(x)}{\partial \theta} - \sum_{i=1}^{m} \sum_{x_i \in \text{false}} n_i(x) \frac{\partial \phi_i(x)}{\partial \theta} \]

\[ n_i(x) \frac{\partial \phi_i(x)}{\partial \theta} \text{ for } x_i \in \text{true} \]

\[ -n_i(x) \frac{\partial \phi_i(x)}{\partial \theta} \text{ for } x_i \in \text{false} \]

\[ \text{for } i = 1, \ldots, m \]
with respect to its weight is

$$\frac{\partial}{\partial w_i} \log P_w(X = x) = n_i(x) - \sum_{x'} P_w(X = x') n_i(x')$$

where the sum is over all possible databases $x'$, and $P_w(X = x')$ is $P(X = x')$ as computed using the current weight vector. Intuitively, the $i$ component of the gradient is the difference between the number of true groundings of the $i$th formula and its expectation according to the current model.

However, this method quickly becomes intractable. First, counting the number of true groundings of a formula in a database is $\#P$-complete. Second, computing the expected number of true groundings is intractable, since it requires exact inference over the model.

If the domain is large enough that the first problem causes learning to be intractable, we approximate the number of true groundings of a formula rather than counting them. To do so, we uniformly sample groundings of the formula, and take the proportion of true samples of the samples as the proportion of true groundings of the formula.

To deal with the second problem, we maximize over the pseudo-log-likelihood of the data, rather than the log-likelihood. If $x$ is a possible world and $x_l$ is the truth value of the $l$th ground atom, the pseudo-log-likelihood of $x$ given weights $w$ is

$$\log P_w^*(X = x) = \sum_{l=1}^n \log P_w(X_l = x_l \mid MB_x(X_l))$$

where $MB_x(X_l)$ is the state of the Markov blanket of $X_l$ in $x$. The Markov blanket of a node $X_l$ in a Markov network is the set of nodes that neighbor $X_l$. The gradient of the pseudo-log-likelihood is

$$\frac{\partial}{\partial w_i} \log P_w^*(X = x) = \sum_{l=1}^n \left( n_i(x) - P_w(X_l = 0 \mid MB_x(X_l)) n_i(x_{[X_l=0]}) ight)$$

$$- P_w(X_l = 1 \mid MB_x(X_l)) n_i(x_{[X_l=1]})$$

where $n_i(x_{[X_l=0]})$ is the number of true groundings of the $i$th formula when we force $X_l = 0$ and leave the remaining data unchanged. Likewise for $n_i(x_{[X_l=0]})$. Computing this gradient does not require inference over the model, and is therefore much more efficient than computing the gradient of the log-likelihood.

We can make the calculation more efficient with a couple simple optimizations. First, the sum in the previous equation can be made more efficient by ignoring predicates that do not appear in formula $i$. Second, the counts $n_i(x)$, $n_i(x_{[X_l=0]})$, and $n_i(x_{[X_l=1]})$ do not change from iteration to iteration, so they need only be computed once and the value stored. Finally, we can ignore ground formulas whose truth value is robust against change of the truth value of any single literal. Such ground formulas are those with at least two true literals. This technique is analogous to the two-literal watch scheme employed by most DPLL-based SAT solvers (Section 1.2.2).

With these optimizations in place, we use an optimizer, such as L-BFGS [25, 56]. We are left with an efficient method for generative weight learning for Markov logic networks.
2.4.2 Discriminative Weight Learning

Unlike generative weight learning, in discriminative weight learning we learn the distribution of some query predicates conditioned on a set of evidence predicates. If we let \( Y \) be the set of ground query atoms, and \( X \) be the set of ground evidence predicates, the conditional likelihood of \( Y \) given \( X \) is

\[
P(Y = y | X = x) = \frac{1}{Z(x)} \exp \left\{ \sum_i w_i n_i(x, y) \right\}
\]

where \( n_i(x, y) \) is the number of true groundings of the \( i \)th formula in the data, and \( Z(x) \) is the partition function. Note that since this is the conditional likelihood, the partition function is parameterized by \( x \), since we do not consider worlds in which the evidence predicates have the opposite truth values of those that appear in the database. Furthermore, we can then ignore any ground formulas satisfied by an evidence atom as again, the truth values of evidence atoms do not change.

To learn weights discriminatively we maximize the conditional log-likelihood (CLL). Because the truth values of evidence atoms do not change, the set of possible states is much smaller for optimization of the CLL. As a result, discriminative weight learning can be done far more efficiently than generative learning, even when using the pseudo-likelihood \[58\]. Many methods exist for optimizing the CLL. We present a few below.

**Voted Perceptron** Gradient descent algorithms update a weight vector \( w \) on each step by using the gradient \( \nabla \) scaled by a learning rate \( \gamma \):

\[
w_{t+1} = w_t - \gamma \nabla
\]

For an MLN, the gradient is the derivative of the (negative, since most optimization algorithms model minimization) CLL. The derivative of the negative CLL with respect to a weight is the difference of the expected number of true groundings of a clause and the actual number of true groundings of that clause according to the data:

\[
\frac{\partial}{\partial w_i} \left( -\log P_w(Y = y | X = x) \right) = E_{w,y}[n_i(x, y)] - n_i(x, y)
\]

\[
= -n_i(x, y) + \sum_{y'} P_w(Y = y' | X = x) n_i(x, y')
\]

where \( x \) is the state of the evidence atoms, \( y \) is the state of the non-evidence atoms in the data, and \( E_{w,y} \) is the expectation over the non-evidence atoms \( Y \). The expectation depends on the current weight vector \( w \), and summed over the states of the world \( y \). We use the notation in \[15\] and let \( E_w[n_i] \) be shorthand for \( E_{w,y}[n_i(x, y)] \).

Computing the expectation \( E_w[n_i] \) is intractable, since it requires summing over possible worlds. The idea of the voted perceptron (VP) algorithm \[9\] is to approximate the \( E_w[n_i] \) with the counts in the most probable state. To avoid overfitting, VP returns the average of the weights on all of the iterations, rather than the final weight vector of the gradient descent. We use MaxWalkSAT or some other MaxSAT solver to perform MPE inference on each iteration to find the most probable state.
**Contrastive Divergence** While VP approximates the expectation by using the most probable state, *contrastive divergence* (CD) does so by sampling states using an MCMC method [27]. While Gibbs sampling is typically used with CD, for MLNs we have the alternative of using MC-SAT to obtain the samples. Since MC-SAT obtains samples that are less closely related than a sweep of Gibbs samples, using MC-SAT provides a better approximation of the gradient with fewer samples.

**Scaled Conjugate Gradient** The last two techniques were used to approximate the gradient, making a gradient-descent algorithm feasible for use. Scaled Conjugate Gradient Descent speeds up the gradient descent process itself, independent of how the gradient is computed. It is typical for gradient descent algorithms to use line search to find the optimal point along a gradient, rather than using a fixed learning rate. However, on certain surfaces this can cause a great deal of “zig-zagging”, making the search less efficient. To combat this, we can impose what is called a *conjugate gradient* [57], in which we impose the condition that the gradient along previous directions must remain zero. This works well, except that doing a line search for MLNs involves computing the objective function, which involves computing the partition function, which is intractable. To deal with this, we use the Hessian matrix, a matrix of partial second derivatives, instead of a line search to choose the step size. This method is known as *scaled conjugate gradient* (SCG) [43]. Lowd and Domingos [37] found this method to be most effective for discriminative learning for MLNs.

### 2.5 Related Work

In this section we briefly discuss a few other systems for statistical relational learning. We then discuss how Markov logic networks subsume many of them, and can therefore be thought of as an interface layer for artificial intelligence. For a more in-depth look at other SRL systems and how they compare, we refer the reader to [31, 24].

*Probabilistic relational models* (PRMs) [23] are a representation scheme for statistical models — one of the first successful SRL methods. A PRM can be thought of as a Bayesian network extended with the concept of domain objects. Each node in the Bayesian network can be thought of as a table in a relational database. An edge, then, exists from node $v$ to $u$ if there is a dependency between attributes from the different tables.

*Relational dependency networks* (RDNs) [48] approximate a joint distribution with a bidirected graph, where nodes are conditional probability tables. Two variables are conditionally independent if they are in separate subgraphs. Like with generative learning in MLNs, RDNs use the pseudo-likelihood learning algorithm to estimate the joint distribution.

*Bayesian logic programs* (BLPs) [30] are an SRL framework that extends Bayesian networks with logic programming [45]. As an MLN is a FOL template for the creation of Markov networks, a Bayesian logic program is a logical program template for the creation of Bayesian networks. A logic program is a set of clauses of the form $A : B_1, \ldots, B_n$, where $A$ and $B_i$ are universally quantified atoms, and $A$ is true iff the $B_i$s are entailed. BLPs are considered one of the more successful SRL schemes.

*Probabilistic soft logic* (PSL) [5] is a weighted logic language that only accepts constraints of the form $H \leftarrow \mu B_1 \land \cdots \land B_n$, where $H$ and $B_i$ are all atoms, and $\mu$ is a real number of infinity, and represents the weight of $H$ being true given that $B_1 \land \cdots \land B_n$ is true. PSL is not as expressive as Markov logic networks or other SRL systems, however, MPE inference can be performed in polynomial time.
*Bayesian Logic*, or BLOG, is an SRL system for working with domains with an unknown number of objects. BLOG models consist of rules that describe look like a mix of C syntax and formal logic. BLOG models can also reason about continuous domains, such as $x$-$y$ coordinates. Since its introduction, MCMC inference methods have been developed for BLOG models.
Chapter 3

Hybrid Markov Logic — a Proposed Framework

Our purpose is to propose an SMT-based framework for hybrid Markov logic networks. In the first section of this chapter, we outline previous work done in defining and implementing hybrid MLNs. In the second section we give our extended definition of hybrid MLNs, and in the subsequent chapters propose inference and learning algorithms for this extended hybrid Markov logic.

3.1 Hybrid Markov Logic — State of the Art

In their 2008 paper [60], Wang and Domingos introduce hybrid Markov logic networks (HMLNs) as an extension of MLNs to the continuous domains. Other than a summarization of this work in [15], no further work has been published about HMLNs. In this section we give their formalism for HMLNs, and discuss the algorithms they presented for MPE and conditional inference.

3.1.1 Definition

As defined in [60], hybrid Markov logic networks are an extension of MLNs that deal with numeric variables and continuous domains. To perform this extension, we modify the definition of an MLN in two ways. First, we allow numeric properties of objects to be nodes in Markov network. A numeric property is a function of one or more objects that maps to a numeric value. For example, if we have a domain type \texttt{person}, a numeric property might be \texttt{Height(person)}. Second, in addition to having weighted FOL features, we also have numeric terms as features. A numeric term is a numeric expression whose value is contributed to the weight of the model.

Here we give the formal definition given by Wang and Domingos in [60]:

\textbf{Definition 3.1.} A hybrid Markov logic network \(L\) is a set of pairs \((\varphi_i, w_i)\), where \(\varphi_i\) is a first-order logic formula or a numeric term, and \(w_i\) is a real number. Together with a finite set of constants \(C = \{c_1, c_2, \ldots, c_{|C|}\}\), it defines a Markov network \(M_{L,C}\) as follows:

1. \(M_{L,C}\) contains one node for each possible grounding with constants in \(C\) of each predicate or numeric property appearing in \(L\). The value of a predicate node is 1 if the ground predicate is
true, and 0 otherwise. The value of a numeric node is the value of the corresponding ground term.

2. $M_{L,C}$ contains one feature for each possible grounding with constraints in $C$ of each formula or numeric term $F_i$ in $L$. The value of a formula feature is 1 if the ground formula is true, and 0 otherwise. The value of a numeric feature is the value of the corresponding ground term. The weight of the feature is the $w_i$ associated with $\varphi_i$ in $L$.

Note that this formalism does not allow for manipulating numeric variables directly. However, this formalism subsumes such a feature: simply include a type numeric in your HMLN, and have only one numeric property Value(numeric) that operates on it.

The authors provide a few abstractions to make performing desirable tasks on HMLNs easier. The first is that in a numeric term, a Boolean predicate is treated as an indicator function. This makes it easy to say, for example, “Add the negative length of $D$ to the weight of the world if $D$ is part of the shortest route,” by adding the constraint $\text{InPath}(D) \cdot \text{Length}(D)$ with weight -1.0.

The second extension they add is the notion of soft equality. Without the addition of numeric terms, we can give certain weight to two numeric objects being equal by adding axioms for equality, so we could say with weight $w$, $\text{Value}(x) = \text{Value}(y)$. However, with HMLNs we can express not just the notion that we prefer two objects to be equal, but that we prefer close values to far values. A numeric term $-(\alpha - \beta)^2$, where $\alpha$ and $\beta$ are numeric terms, gives a Gaussian penalty for diverging from equality, where the standard deviation is $1/\sqrt{2w}$. We will use the notation $\alpha \approx_G \beta$, to indicate this kind of Gaussian soft equality constraint.

The last extension, soft inequality, is similar to the second. The shorthand $\alpha \succ t$, where $\alpha$ is a numeric term and $t$ is a numeric constant is used for the expression $-\log(1 + e^{a(t - \alpha)})$, where $a$ is a predetermined constant. Stated otherwise, the (log) sigmoid function is used to represent soft inequality, meaning that as the values get closer, the weight increases. Similarly, we have $\alpha \prec t$ as shorthand for $-\log(1 + e^{a(t - \alpha)})$.

### 3.1.2 MPE Inference in HMLNs

Wang and Domingos put forth Hybrid MaxWalkSAT as an algorithm for MPE inference in HMLNs [60]. It works like MaxWalkSAT, but invokes L-BFGS (or, in principle, some other numerical optimization algorithm) as a subroutine. Like WalkSAT, with probability $p$ it takes a random step. This means selecting a random variable. If the variable is Boolean, it is simply flipped. If it is numeric, then that variable is optimized using L-BFGS, and Gaussian noise is added to it. With probability $1 - p$, a greedy step is made. This means performing a one-dimensional optimization with each variable, and making the change to the variable that causes the largest net improvement in the total weight. Pseudocode for Hybrid MaxWalkSAT is given in Algorithm 3. In the algorithm, $x$ represents an assignment of variables, and $S(x)$ represents the score or weight of that assignment. Calls to a numerical optimizer are made in lines 13 and 26.

In their experiments, Wang and Domingos found that Hybrid MaxWalkSAT outperformed simulated annealing by about two orders of magnitude. However, experiments were only run on a single HMLN, so more evidence is needed to be sure of the algorithm’s efficacy.

### 3.1.3 Conditional Inference in HMLNs

Wang and Domingos present a modified version of the MC-SAT algorithm presented in Section 2.3.2 for use with HMLNs. As MC-SAT uses a SAT sampling program, e.g. SampleSAT, and a SAT Solver,
Algorithm 3 Hybrid MaxWalkSAT

1: procedure HMaxWalkSAT(clauses, numeric terms, weights, max_tries, max_flips)
2: \( x^* \leftarrow \text{null} \)
3: \( S(x^*) \leftarrow -\infty \)
4: for \( i \) from 1 to max_tries do
5: \( x \leftarrow \text{random assignment} \)
6: for \( j \) from 1 to max_flips do
7: if \( S(x) > S(x^*) \) then
8: \( x^* \leftarrow x \)
9: \( S(x^*) \leftarrow S(x) \)
10: end if
11: \( c \leftarrow \text{random unsatisfied clause or numeric term} \)
12: if \( \text{uniform}(0, 1) < p \) then
13: \( x \leftarrow \arg \max_x(c) \)
14: if \( x \) is numeric then
15: \( x \leftarrow x + \text{Gaussian noise} \)
16: end if
17: else
18: for each variable \( x_i \) in \( x \) do
19: \( x'_i \leftarrow \arg \max_{x_i} S(x) \)
20: \( S(x'_i) \leftarrow S(x) \) with \( x_i = x'_i \)
21: end for
22: \( x'_f \leftarrow x_i \) with highest \( S(x'_i) \)
23: if \( S(x'_f) > S(x) \) or \( c \) is a clause then
24: \( x_f \leftarrow x'_f \)
25: else
26: \( x_c \leftarrow \arg \max_{x_c} S(x) \), where \( x_c \) is the subset of \( x \) appearing in \( c \).
27: end if
28: end if
29: end for
30: end for
31: return \( x^* \)
32: end procedure
e.g. WalkSAT, so does hybrid MC-SAT use a hybrid SAT sampler and a hybrid SAT solver.

Hybrid WalkSAT (HWS) is an extension of WalkSAT that includes constraints of the form $f_k(x) \geq a$ in addition to clauses. Given a set of such numeric constraints and clauses, HWS attempts to find the assignment that maximizes the number of satisfied constraints. HWS works similarly to HMWS, except that 1) since the numerical constraints are bounded, each step maximization of $f_k(x)$ is halted as soon as $f_k(x) \geq a$, and 2) $S(x)$ is the number of satisfied constraints instead of the sum of the weights of the features.

Hybrid SampleSAT (HSS) is a program for generating uniformly random samples of assignments that satisfy a set of constraints $M$. It works by alternating HWS and simulated annealing steps. A new candidate state is chosen by choosing a random variable, and flipping it if it is Boolean or adding Gaussian noise if it is numeric.

Hybrid MC-SAT (HMCS) Like MC-SAT, one auxiliary variable $u_k$ is introduced for each feature $f_k$. The auxiliary variable $u_k$ is sampled uniformly from $[0, e^{w_k f_k(x)}]$. Boolean features are treated the same as in MC-SAT, and for numeric feature $f_k$, the standard slice sampling constraint $f_k > \log(u_k)/w_k$ is used. Slices are sampled using HSS.

HMCS was tested against Gibbs sampling and simulated tempering on the same benchmark used for Hybrid MaxWalkSAT, and outperformed those methods. Again, since only one HMLN was used, more experiments need to be run to determine how efficacious HMCS is compared to other methods.

In the subsequent sections, we give an augmented formulation of HMLNs, and propose some methods for learning and inference within that framework.

### 3.2 A New Formulation for Hybrid Markov Logic Networks

While the formulation for hybrid Markov logic networks (HMLNs) given in [60] successfully extends MLNs to the continuous domain with the use of numeric variables. However, it does not allow us to create models with hard constraints that use the numeric variables.

As we saw with SAT, we sometimes find expressing certain models in MLNs to be a bit cumbersome. Just as SMT was developed to make modeling certain domains more expressive, and in certain cases more efficient, we propose a formulation of HMLNs that allows for background theories including uninterpreted functions, arithmetic, and other theories supported by modern SMT solvers.

In this section we give our new formulation explicitly, then propose some algorithms for inference and learning with these HMLNs.

#### 3.2.1 An SMT-like Formulation for Hybrid Markov Logic Networks

Here we give a formal definition for HMLNs that is an extension of the one given in Section 3.1.1. In addition to numeric terms as features, we also allow background theories as implemented in SMT solvers for use with formula constraints. For example, if we are using an SMT engine that implements linear real arithmetic, then we can use the associated predicate $\leq$. We call this formalism augmented hybrid Markov logic networks (AHMLNs), and define it as follows:

**Definition 3.2.** A hybrid Markov logic network $L^T$ is a set of pairs $(\varphi_i, w_i)$, where $\varphi_i$ is a first-order logic $T$-formula or a numeric term, and $w_i$ is a real number. Together with a finite set of constants $C = \{c_1, c_2, \ldots, c_C\}$, it defines a Markov network $M_{L,C}$ as follows:
1. **MLC contains one node for each possible grounding with constants in C of each predicate or numeric property appearing in L.** The value of a predicate node is 1 if the ground predicate is true, and 0 otherwise. The value of a numeric node is the value of the corresponding ground term.

2. **MLC contains one feature for each possible grounding with constraints in C of each formula or numeric term $F_i$ in L.** The value of a formula feature is 1 if the ground formula is true, and 0 otherwise. The value of a numeric feature is the value of the corresponding ground term. The weight of the feature is the $w_i$ associated with $\varphi_i$ in L.

By allowing the Boolean constraints to be $T$-formulas, we greatly augment the expressive power of the HMLN framework. We next propose algorithms for inference and learning in this AHMLN framework.

### 3.2.2 MPE Inference in AHMLNs

The problem of finding the most probable explanation in AHMLNs is very similar to that in HMLNs. The only difference is that the clausal constraints can contain $T$-literals. We start by noting that if all of the constraints are $T$-clauses, that is there are no numeric term constraints, then the problem MPE inference in AHMLNs reduces to partial weighted MaxSMT.

**MPE Inference with Lemma-lifting** We extend the lemma-lifting algorithm given in Section 1.3.5 to handle numeric term constraints. The lemma-lifting algorithm works by repeatedly getting the MPE of the Boolean abstraction if the formula, and then checking its $T$-consistency, returning learned $T$-lemmas. The MPE is obtained through the use of a standard MaxSAT solver.

Modifying it to work for AHMLNs is quite simple — we replace the calls to a MaxSAT solver with calls to HMWS. On each iteration, HMWS returns the MPE of the Boolean abstraction of the AHMLN, along with assignments to all of the numeric terms that optimize the numeric term constraints. Then, the SMT solver checks the consistency of the assignment to the numerical variables, returning any $T$-lemmas it learns.

### 3.2.3 Conditional Inference in AHMLNs

Ermon et. al. give an algorithm for sampling over a high-dimensional discrete set, such as a graphical model in [19]. We present the algorithm, describe how it can be used for conditional inference in MLNs, and then describe how to use it for conditional inference in AHMLNs.

**PAWS: Discrete Sampling with Universal Hashing** Suppose we want to sample from a probability distribution defined over an undirected graphical model

$$p(x) = \frac{1}{Z} w(x) = \frac{1}{Z} \prod_{\alpha \in I} \psi_{\alpha}(\{x\}_\alpha)$$

where $\{x\}_\alpha$ is an assignment of the subset of the variables and $\psi_{\alpha} : \{x\}_\alpha \rightarrow \mathbb{R}^+$ are features of the distribution. The algorithm put forward in [19] works in three steps. The first step is to, given the distribution $p$, create an approximate distribution $p'$ that is close to $p$ but is more discrete. The second step is to define a uniform probability distribution $p''$ from $p'$ over a higher-dimensional
embedding of $X = \{0, 1\}^n$. Finally, we sample uniformly from $p''$, then project the embedding on a lower-dimensional space and search for feasible states.

The first step is to discretize the distribution $p$. In order to do so, we first calculate the highest weight possible in the distribution, i.e. find the MPE. Once we have $M$, the maximum probability of universe, we create $l + 1$ disjoint buckets $B_i = \{x \mid e(x) \in (\frac{M}{2^i+1}, \frac{M}{2^i})\}, i = 0, \ldots, l - 1$ and $B_l = \{x \mid w(x) \in [0, \frac{M}{2^l}]\}$. Intuitively, we create $l + 1$ buckets with values that are exponentially related. For a given input $x$, $w'(x)$ is equal to the closest bucket to $w(x)$.

The second step is embed this discretized distribution into a higher-dimensional space so as to reduce the problem of sampling from $p'$ to the problem of uniform sampling. To do so we embed $p'$ in the higher-dimensional space $p''$. We give here a simplified, but potentially less efficient embedding than that used in [19]. The embedding is $S(w, l)$ where $w$ is the weight function of the discreted distribution $p'$, and $l$ is the number of buckets.

$$S(w, l) = \{ (x, y_1, y_2, \ldots, y_{l-1}) \mid w(x) \leq \frac{M}{2^i} \implies y_i, 1 \leq i \leq l - 1; \ w(x) > \frac{M}{2^{l-1}} \}$$

If this embedding is sampled uniformly, the resulting samples will be according to the distribution $p'$. This can be seen from the implication constraints. If $y_i$ is 0, then we must be sampling from a bucket larger than the $i$th one. Since the $y$ variables are binary, and since the buckets are exponentially related, we sample according to the distribution $p'$.

The final step is to sample $p''$ uniformly. The authors present a method very similar to the method of XOR Sampling introduced in [20] and described in Section 1.2.4, $k$ random XOR constraints are added to the distribution. Then a model counter is used to enumerate the remaining satisfying solutions. If fewer than $P$ configurations remain, we choose one by rejection sampling. Ermon et. al. give a detailed proof of the correctness of the algorithm. The samples can be made arbitrarily close to the distribution $p$ by tweaking various parameters of the algorithm.

The PAWs algorithm was testing on Ising models and software verification test generation models against belief propagation, Gibbs sampling, and the authors’ previous contribution, the WISH algorithm [18]. PAWS significantly outperformed the other algorithms on both test problems.

PAWs for conditional inference in MLNs

The motivating example for PAWs is an undirected graphical model. Since ground MLNs are simply specialized undirected graphical models, using PAWs for conditional inference in MLNs is straightforward.

To find the maximum weight, we simply invoke an MPE algorithm, such as MaxWalkSAT. Next, we create the discretized distribution according to the parameters given to PAWS and the weight of the MPE. Performing the embedding is straightforward: we add $l - 1$ new binary variables, and add constraints that set those variables based on the sums of the weights. Although a more efficient solution probably exists, we can simply use SMT constraints to accomplish this task. Finally, we sample with the modified XORSample algorithm. Since we used SMT constraints in the embedding, we will have to use SMT and MaxSMT solvers where SAT and MaxSAT solvers would be used, respectively. Alternatively, we could compile away the SMT constraints in the manner that an eager SMT solver would. After obtaining a sufficient number of samples, the conditional probability of the query predicate is simply the proportion of samples in which the query predicate is true.

PAWS for conditional inference in AHMLNs

Extending the algorithm to work for AHMLNs is straightforward. We use the AHMLN MPE algorithm described above to get $M$. To perform the embedding, we simply add the numeric term constraints to the sums in the SMT constraints used
above. Again, when performing the uniform sampling we use SMT and MaxSMT solvers where appropriate.
Bibliography


37


