Algorithms for Markov Logic Networks

José Francisco Pires Amaral

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Vasco Miguel Gomes Nunes Manquinho

Examination Committee

Chairperson: Prof. Miguel Nuno Dias Alves Pupo Correia
Supervisor: Prof. Vasco Miguel Gomes Nunes Manquinho
Member of the Committee: Prof. Francisco António Chaves Saraiva de Melo

November 2017

Acknowledgments

This work was partially supported by national funds through FCT with reference UID/CEC/50021/2013.
I would like to thank the supervisor of this thesis, Professor Vasco Miguel Gomes Nunes Manquinho for his dedication, effort and share of knowledge. I want to thank Professor Mikoláš Janota for his assistance regarding the use and improvement of the Satisfiability Modulo Theories (SMT) solver calls.

Resumo

Muitas aplicações do mundo real têm que lidar com incerteza. Assim, estas aplicações requerem métodos probabilísticos para representar informação do domínio de modo a descrever a sua probabilidade. Além disso, estas aplicações são normalmente muito grandes e necessitam de uma forma compacta para representar informação. Por exemplo, lógica de primeira ordem não só permite o uso de frases que contêm variáveis, mas também o uso de quantificadores universais e existenciais. As Redes Lógicas de Markov (MLN) combinam ambos estes tópicos. As MLN associam fórmulas em lógica de primeira ordem a um valor real (peso) que representa a sua probabilidade.

Esta tese investiga duas hipóteses. Em primeiro lugar, exploramos o uso de Satisfação Máxima incremental para resolver problemas de MLN. Métodos existentes que resolvem MLN dependem de várias chamadas independentes a uma ferramenta que resolve problemas de Satisfação parcial máxima com pesos (MaxSAT). O número de chamadas é desconhecido no início, mas normalmente depende do tamanho e da estrutura do problema. O nosso objectivo principal é tirar proveito das recentes melhorias nas técnicas para resolver MaxSAT. Neste caso particular, queremos que as anteriores chamadas à ferramenta de MaxSAT sejam úteis para as chamadas seguintes. As soluções do estado da arte fazem com que cada chamada à ferramenta MaxSAT comece a procura sempre do início enquanto que as nossas chamadas usam informação obtida nas iterações anteriores, convergindo assim mais rapidamente para uma solução.

Em segundo lugar, exploramos a ideia de usar uma ferramenta que resolve problemas de Satisfação Módulo Teorias (SMT) para validar cada uma das regras. Ou seja, existe uma fase em que o algoritmo necessita de verificar se a solução actual falsifica ou não as regras ainda não exploradas. Algumas abordagens do estado da arte usam uma ferramenta de *Datalog* para validar as regras. Este método implica o uso de uma base de dados. Um dos nossos objectivos é evitar o uso de uma base de dados. O uso de uma ferramenta SMT permite-nos validar cada uma das regras sem inquirir uma base de dados, poupando assim algum tempo por não ter os atrasos inatos a esta chamada.

No final, avaliamos a nossa abordagem com duas ferramentas do estado da arte (Tuffy [37] e IPR [26]). Por um lado, o nosso algoritmo, Inference using PROpagation and Validation (ImPROV) devolve soluções correctas e óptimas, que era o nosso principal objectivo. Já que o Tuffy não garante soluções correctas nem óptimas, a nossa abordagem é melhor que a do Tuffy. Por outro lado, o algoritmo Inference via Proof and Refutation (IPR), que garante soluções correctas e óptimas, tem um melhor desempenho (tempo total de computação) do que o nosso algoritmo. Nas instâncias utilizadas, o IPR devolve soluções correctas e óptimas com mais rapidez que o ImPROV.

Palavras-chave: Redes lógicas de Markov, Satisfação Módulo Teorias, Satisfação Máxima.

// Abstract

Many real world applications have to deal with uncertainty. Consequently, they demand probabilistic methods to represent some (if not all) of their events to describe their likelihood. Moreover, these applications are usually large-scale and need a compact and plain representation: first-order logic. First-order logic not only allows the use of sentences that contain variables but also the use of existential and universal quantifiers. Markov Logic Network (MLN) combines both these aspects. MLN associates first-order logic formulas to a real value (weight) that represents their likelihood.

This thesis explores two main hypothesis. First, we explore the use of incremental Maximum Satisfiability (MaxSAT) to solve MLN problems. Existing methods that solve MLN rely on various independent Weighted Partial Maximum Satisfiability (WPMS) solver calls. The number of calls is unknown beforehand but usually depends on the size and the difficulty of the problem. Our main goal is to benefit from recent improvements in WPMS solving techniques. In this particular case, we intend to make previous WPMS solver calls useful to the subsequent WPMS solver calls. State-of-the-art approaches make each WPMS call start the search from the beginning whereas our WPMS solver calls use previously learned information, thus converging faster to the solution.

Secondly, we explore the hypothesis of using an Satisfiability Modulo Theories (SMT) solver to validate the constraints. In other words, during the algorithm we need to guarantee that the current solution does not violate any rules that are not yet instantiated. Some state-of-the-art approaches use a Datalog solver to validate each constraint. This method implies the use of queries to a database. Our main goal is to avoid having a database altogether. Using an SMT solver allows us to validate rules without querying a database, therefore saving some time from the attached delays of using a database.

Finally, we evaluate our approach with two state-of-the-art tools (Tuffy [37] and IPR [26]). On the one hand, our algorithm, Inference using PROpagation and Validation (ImPROV) returns sound and optimal solutions, which is our first and most important goal. Since Tuffy does not guarantee correct solutions (neither sound nor optimal), our approach is better than Tuffy's. On the other hand, the Inference via Proof and Refutation (IPR) algorithm, which guarantees sound and optimal solutions, has a better performance (total computation time) than our algorithm. IPR returns correct solutions faster than ImPROV.

Keywords: Markov logic networks, Satisfiability Modulo Theories, Maximum Satisfiability.

Contents

Acknowledgments iii

Resumo v

Abstract vii

List of Algorithms xi

List of Figures xiii

List of Tables xv

1 Introduction 1

2 Preliminaries 5

2.1 Propositional Satisfiability 5

2.2 Maximum Satisfiability 5

2.3 Markov Logic Networks 6

2.4 Satisfiability Modulo Theories 9

2.5 Overview 9

3 Algorithms for MLN 11

3.1 Grounding techniques 11

3.2 Algorithms for Maximum Satisfiability (MaxSAT) 13

3.3 Algorithms for MLN 18

4 ImPROV (Inference using PROpagation and Validation) 23

4.1 Architecture 23

4.2 Algorithms 24

4.2.1 Algorithmic structure 25

4.2.2 Preprocessing and Initializations 26

4.2.3 Rule validation 27

4.2.4 Incremental MaxSAT 30

5 Evaluation 33

5.1 MLN Instances 33

5.2 ImPROV's configuration 34

5.2.1 Initial eager propagation 34

5.2.2 Counterexample generation 35
5.2.3 Incremental vs non-incremental MaxSAT 39
5.2.4 Results with ImPROV's best configuration 39
5.3 Analysis of the ImPROV algorithm 40
5.3.1 Correctness of the algorithms 40
5.3.2 Number of iterations 41
5.3.3 Number of grounded clauses 41
5.3.4 Solution cost 42
5.3.5 Performance of the algorithms 42
5.4 Summary 43

6 Conclusions **45**

Bibliography **53**

List of Algorithms

3.1 Relax Clauses Function 13

3.2 Linear Search Sat-Unsat Algorithm 14

3.3 Linear Search Unsat-Sat Algorithm 15

3.4 The WMSU3 Algorithm [29] 16

3.5 Fu-Malik Algorithm with Incremental SAT [30] 17

3.6 Cutting Plane Inference (CPI) Algorithm 19

3.7 Soft-Cegar Algorithm 20

3.8 IPR Algorithm 21

4.1 ImPROV Algorithm 26

List of Figures

2.1 An MLN for scheduling classes including evidence. 7
2.2 Ground Markov Random Field (MRF) achieved by using the formulas and the evidence presented previously. 8

3.1 Full grounding of the MLN presented in figure 2.1 (clausal form). 12

4.1 Software architecture. 24
4.2 Example of an MLN program. 24
4.3 Example of an evidence file and a query file. 25
4.4 Solution for MLN defined in figures 4.2 and 4.3. 25
4.5 SMT encoding when validating the third rule. (first method) 28
4.6 SMT encoding when validating the third rule. (second method) 28

List of Tables

2.1 Example of a conversion from first-order logic to clausal form. F() is short for Friends(), T() for Teaches(), L() for Likes(), A() for Attends() and P() for Popular(). 8

5.1 Statistics of applications' constraints and databases. (K = thousand, M = million) 34

5.2 Results of ImPROV with and without the initial eager propagation phase. The instances used are the ones presented in table 5.1. (m = minutes, s = seconds) 35

5.3 Results of ImPROV using a different number of counterexamples returned by the validation call. Experiments that timed out (denoted '-') exceeded the time limit. The instances used are the ones presented in table 5.1. (m = minutes, s = seconds) 37

5.4 Times obtained with ImPROV using the best possible configuration. The instances used are the ones presented in table 5.1. (m = minutes, s = seconds) 39

5.5 Results of evaluating TUFFY, IPR and ImPROV on three benchmark applications. Experiments that timed out (denoted '-') exceeded the time limit. The instances used are the ones presented in table 5.1. 41

Chapter 1

Introduction

Many real world applications or problems demand probabilistic methods to depict some (if not all) of their events to either mark their importance among the group or describe their likelihood. These real world applications are usually large-scale and need a compact and plain representation: first-order logic. First-order logic not only permits the use of existential and universal quantifiers, as well as predicates or relations.

$\forall x$: $Fish(x) \implies Animal(x)$ is a simple example of a first-order logic formula. In this example, we introduce two relations (Fish and Animal) and use an universal quantifier. This formula implies that every fish is an animal.

Markov Logic Network (MLN) combines both these characteristics, i.e. it associates first-order logic formulas to a value (weight) that represents their likelihood. Furthermore, MLN can deal with incomplete or missing evidence, which, in a real life environment, is likely to happen. Usually real world applications do not have clean and complete data, making this characteristic of MLN useful in most cases. MLN combines both these characteristics without any constraints except the finiteness of the domain [41].

There are already several practical examples of Markov Logic Network (MLN). Chahuara et al. [5] present the possibility of recognizing Activities of Daily Living (ADL) in a smart home. The obtained results showed that the use of MLN was significantly more accurate than the other tested methods: Support Vector Machine (SVM) [7] and Naive Bayes (NB). Dierkes et al. [11] use MLN in order to estimate the effect of word of mouth on churn and cross-buying in the mobile phone market. Their results provided evidence that word of mouth has a substantial impact on the customers' decisions. Snidaro et al. [50] introduce the use of MLN to further develop anomaly detectors or event recognition systems for maritime situational awareness. In this case, the use of MLN gives the system the ability to reason with incomplete or missing evidence, a common feature in maritime domain. Advisor Recommendation (AR) [1], also known as Link prediction, Entity Resolution (ER) [49], Information Extraction (IE) [39], Program Analysis (PA) [25] and Relational Classification (RC) [37] are another few of the many possible applications of MLN.

MLN is of good use when there are certain objectives we want to optimize in addition to some rules

[1] The dataset and MLN are available on http://alchemy.cs.washington.edu/

we do not want to break. We seek to find a solution in the fastest and most efficient manner, that is, getting a solution that does not break any of the required conditions and optimizes the objectives of the given problem.

In our proposal, we intend to improve the way some recently proposed algorithms pursue this solution. In order to build our tool, we focused mainly on the algorithmic structure of the Inference via Proof and Refutation (IPR) algorithm presented by Mangal et al. [26]. Our method has a very similar algorithmic structure when compared with IPR. However, there are two key differences during the search:

- Since our problems usually have several iterations, we want to reuse computation throughout the algorithm, i.e. making sure that the tool does not waste its time during the current iteration in order to find something that was already computed beforehand, as well as guaranteeing that the algorithm is using information obtained in the previous iterations to finish the current iteration quicker.
- When looking for counter-examples, we use a Satisfiability Modulo Theories (SMT) solver instead of a Datalog solver. The use of Datalog solver implies the use of a database. Using an SMT solver allows us to validate rules without querying a database. This identification of counter-examples is an important step in our algorithm and it is thoroughly explained later in the document.

By the end of the experiments, our goal is to get better results than the state-of-the-art engines/tools. That is, if these tools are sound and optimal (for instance IPR), we hope to find a solution cost that is equal to the one found by these other engines. On the other hand, if these tools are not sound and optimal (for example Tuffy), we hope to find a solution cost that is either equal or better than the one found by these engines. Since efficiency is very important in this kind of decision problems, our second goal is to find a solution faster than the other state-of-the-art algorithms without losing the soundness and optimality of the algorithm.

Important to note that we solve these problems using a Maximum A Posteriori (MAP) inference method. MAP inference can be used to find the most likely state of world given some evidence. Note that the same problem could be solved using a different inference and therefore have a distinct solution.

In the next chapter we introduce the most important concepts mentioned in the remainder of the document such as Propositional Satisfiability (SAT) problems, Maximum Satisfiability (MaxSAT) problems and Markov Logic Networks (MLN) in a more detailed way. A small and straightforward MLN example is also presented. Chapter 3 briefly overviews the progression and evolution throughout the years on the different grounding techniques, on the distinct possible techniques used to solve MaxSAT problems as well as the most recent methods to solve MLN problems. Several pseudo-codes are demonstrated, including IPR's. Chapter 4 describes each component of the developed tool, its architecture, the algorithms it uses during the search, along with the reasons behind our decisions throughout the developing process. Since our algorithm is based on IPR's structure, we compare both algorithms and the methods used by them. An example from start to finish is also present, as well as the pseudo-code for our algorithm. In chapter 5, we present the results obtained over the tested applications and a comparison

between the described algorithm and two state-of-the-art methods. Moreover, we analyse the obtained results and justify them using our tool's architecture and algorithm. Lastly, chapter 6 concludes with a brief summary of the developed tool, describing its positives, negatives and the results obtained. To finish the document, we present some possible future work that can improve Inference using PROpagation and Validation (ImPROV) and its performance.

Chapter 2

Preliminaries

This chapter describes the most important concepts used throughout the document. We introduce the basics, i.e. how to frame a decision problem and how it is expressed. Then, we present three types of problems: Propositional Satisfiability (SAT) problems, Maximum Satisfiability (MaxSAT) problems and Markov Logic Networks (MLN). As we explain each one of the problems, some small and clear examples are shown, solved and explained. Finally, we briefly present Satisfiability Modulo Theories (SMT).

In propositional logic, formulas are represented over a set of Boolean variables $X = \{x_1, x_2, ..., x_n\}$, where n is the total number of variables and each variable x_i must take one of two values: true or false.

These formulas are usually presented in Conjunctive Normal Form (CNF), that is a conjunction of clauses. Each clause is a disjunction of literals. A literal can be a variable or its negation (e.g. x_i or $\neg x_i$). At least one literal in each clause must be satisfied (have value true) so that the entire formula is satisfied. For instance, $(x_1 \vee x_2) \wedge (x_3 \vee \neg x_4)$ is a formula in CNF with two clauses.

2.1 Propositional Satisfiability

Given a CNF formula ϕ, the goal of Propositional SAT is to find an assignment to the variables in ϕ such that ϕ is satisfied or prove that such assignment does not exist.

For example, $(\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$ is a CNF formula describing a SAT problem. A possible solution is $x_1 = false, x_2 = false, x_3 = false$. Note that there might be more viable solutions for each CNF formula. The one presented is one of the several possible solutions.

On the other hand, there are formulas such that there is no satisfying assignment. For instance, $(x_1) \wedge (x_2) \wedge (\neg x_1 \vee \neg x_2)$ is an unsatisfiable CNF formula. There is not any combination of values for variables x_1 and x_2 that makes the previous CNF formula true.

2.2 Maximum Satisfiability

MaxSAT is an optimization version of SAT. Given a CNF formula ϕ, the goal of MaxSAT is to find an assignment to the variables in ϕ such that it maximizes the number of satisfied clauses. This is

equivalent to minimizing the number of falsified clauses. For the remainder of the document, every MaxSAT problem is considered as a minimization problem.

Consider the unsatisfiable CNF formula $(x_1) \wedge (x_2) \wedge (\neg x_1 \vee \neg x_2)$. A possible MaxSAT solution to this formula is $x_1 = true$, $x_2 = true$, since it falsifies only one clause (third) which is the minimum possible number of falsified clauses with any assignment.

We can represent a conjunction of clauses using set notation. The set of clauses in ϕ can be represented by the union of two sets $(\phi_H \cup \phi_S)$, where ϕ_H is the set of hard clauses and ϕ_S is the set of soft clauses. Hard clauses must be satisfied. A weighted soft clause is a pair (c, w), where c represents a clause and w represents the weight of not satisfying clause c. It is important to note that it is not mandatory to satisfy all soft clauses to have a solution to the MaxSAT problem.

There are some variations of MaxSAT [34]. Weighted MaxSAT occurs when some clauses have more relevance than others. This relevance is represented by its weight (a positive real number). Given a weighted CNF formula, the goal of weighted MaxSAT is to minimize the sum of the weights of the unsatisfied clauses. Consider the previously presented unsatisfiable CNF formula ϕ and the sets $\phi_H = \{\}$ and $\phi_S = \{(x_1, 0.5), (x_2, 2.0), (\neg x_1 \vee \neg x_2, 1.5)\}$, the solution for the weighted MaxSAT problem is $x_1 = false$, $x_2 = true$. In this solution we have an overall cost of 0.5 by not satisfying the first clause $(x_1, 0.5)$. Note that this is the only possible solution to this weighted MaxSAT problem. The only way to have two or more possible and different solutions to the same weighted MaxSAT problem is if these solutions have the exact same weight, i.e. the sum of the weights of the unsatisfied clauses are equal.

The variations partial MaxSAT and weighted partial MaxSAT are extensions of unweighted MaxSAT and weighted MaxSAT respectively.

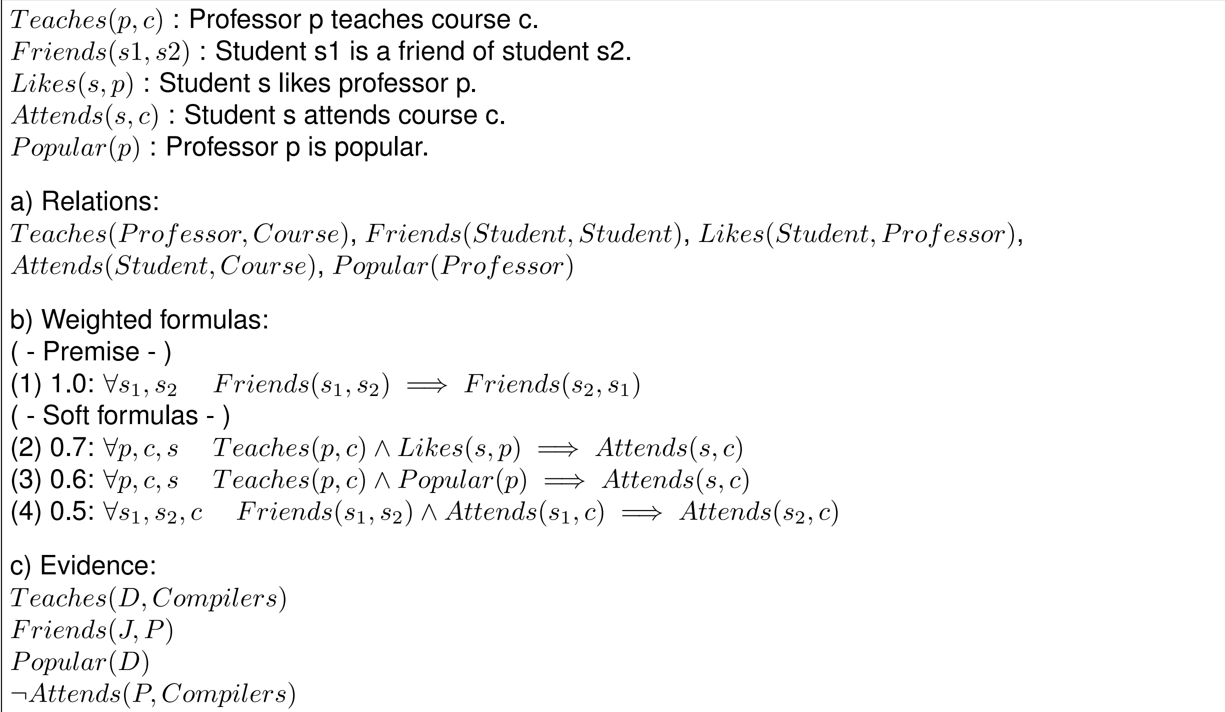
Given the CNF formula $\phi = \phi_H \cup \phi_S$ where $\phi_H = \{(x_1)\}$ and $\phi_S = \{(x_2, 1.0), (\neg x_1 \vee \neg x_2, 1.0)\}$, a solution for the partial MaxSAT problem is $x_1 = true$, $x_2 = false$. This problem is considered a partial MaxSAT problem due to two specific characteristics: first, the set ϕ_H is not empty, that is, there is at least one hard clause; secondly, all soft clauses have the same relevance, the same weight.

Given the same formula ϕ and the sets $\phi_H = \{(x_1)\}$ and $\phi_S = \{(x_2, 2.0), (\neg x_1 \vee \neg x_2, 1.5)\}$, the solution for the weighted partial MaxSAT problem is $x_1 = true$, $x_2 = true$. In this solution we have an overall cost of 1.5 by not satisfying the third clause $(\neg x_1 \vee \neg x_2, 1.5)$. This problem is considered a weighted partial MaxSAT problem since the set ϕ_H is not empty and not all soft clauses have the same relevance, the same weight.

2.3 Markov Logic Networks

Markov Logic Network (MLN) is a simple way to merge both first-order logic and weighted atomic formulas, the latter being atoms with a real number attached to it. An MLN can be defined as a set of weighted first-order logic formulas. A weighted first-order logic formula is a pair (ϕ, w) where ϕ is a first-order logic formula and w is a real number (weight).

Since each first-order logic formula has an attached weight, the notion of hard and soft clauses (presented in the previous section) can be used in MLN. In other words, each first-order logic formula



$Teaches(p, c)$: Professor p teaches course c.
$Friends(s1, s2)$: Student s1 is a friend of student s2.
$Likes(s, p)$: Student s likes professor p.
$Attends(s, c)$: Student s attends course c.
$Popular(p)$: Professor p is popular.

a) Relations:
$Teaches(Professor, Course)$, $Friends(Student, Student)$, $Likes(Student, Professor)$,
$Attends(Student, Course)$, $Popular(Professor)$

b) Weighted formulas:
(- Premise -)
(1) 1.0: $\forall s_1, s_2 \quad Friends(s_1, s_2) \implies Friends(s_2, s_1)$
(- Soft formulas -)
(2) 0.7: $\forall p, c, s \quad Teaches(p, c) \wedge Likes(s, p) \implies Attends(s, c)$
(3) 0.6: $\forall p, c, s \quad Teaches(p, c) \wedge Popular(p) \implies Attends(s, c)$
(4) 0.5: $\forall s_1, s_2, c \quad Friends(s_1, s_2) \wedge Attends(s_1, c) \implies Attends(s_2, c)$

c) Evidence:
$Teaches(D, Compilers)$
$Friends(J, P)$
$Popular(D)$
$\neg Attends(P, Compilers)$

Figure 2.1: An MLN for scheduling classes including evidence.

can be seen as hard or soft.

MLN can also be regarded as a MRF [21] applied to weighted first-order logic. MRF, also known as Markov Network, can be described as an undirected graph where each node is a variable and each edge represents a dependency. It is then possible (depends on the finiteness of the problem) to portray an MLN as an undirected graph where each node is a grounded first-order logic formula (does not contain any free variables) and each edge denotes a relation.

Figure 2.1 shows an abbreviated example of an MLN for scheduling classes [4] where section a) defines the set of relations used in the first-order logic formulas, section b) defines the set of formulas and section c) defines the evidence or facts. The first-order logic formulas are numbered from 1 to 4.

Figure 2.2 represents the undirected graph given by the full grounding of the previous example, where each node is a grounded atom (e.g., $Teaches(D, Compilers)$) and each arc is a dependency that is identified according to the numeration given in figure 2.1. Full grounding is the full instantiation of all facts over all of the relations. This grounding topic is an important step when solving MLN problems and it is further explained in the next chapter.

Each one of these atoms can have a truth value: true or false. Using different combinations of truth values for each node, we are able to create multiple possible world scenarios. In this case, if we ignore the truth values of the evidence given, having ten of these nodes means that there are 1024 possible world scenarios. These numbers show how these logic problems can grow and how enormous they can get.

Regarding the solution of this problem, there is at least one interpretation that minimizes the sum of the falsified soft clauses. One possible optimal (minimum solution cost) solution is to assign the nodes $Teaches(D, Compilers)$, $Friends(J, P)$, $Popular(D)$, $Friends(P, J)$ and $Attends(J, Compilers)$ to true

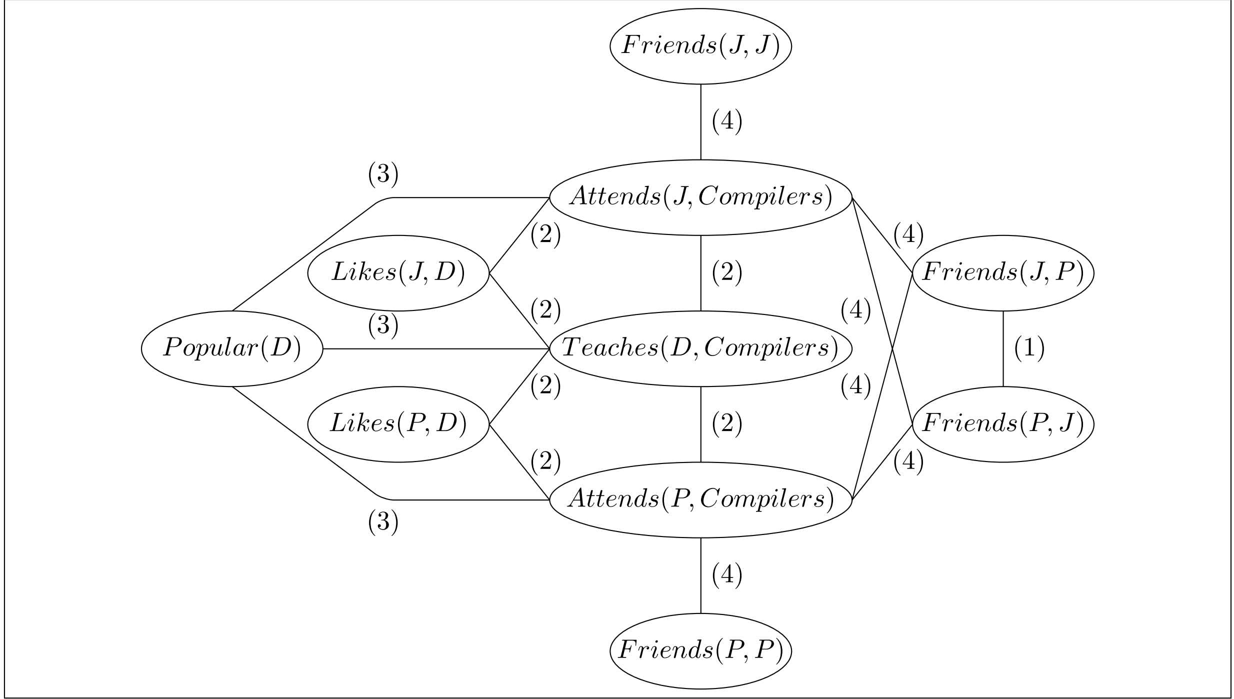


Figure 2.2: Ground MRF achieved by using the formulas and the evidence presented previously.

Table 2.1: Example of a conversion from first-order logic to clausal form. F() is short for Friends(), T() for Teaches(), L() for Likes(), A() for Attends() and P() for Popular().

First-Order Logic	Clausal Form
(1) 1.0: $\forall s_1, s_2 \quad F(s_1, s_2) \implies F(s_2, s_1)$	$\neg F(s_1, s_2) \vee Fr(s_2, s_1)$
(2) 0.7: $\forall p, c, s \quad T(p, c) \wedge L(s, p) \implies A(s, c)$	$\neg T(p, c) \vee \neg L(s, c) \vee A(s, c)$
(3) 0.6: $\forall p, c, s \quad T(p, c) \wedge P(p) \implies A(s, c)$	$\neg T(p, c) \vee \neg P(p) \vee A(s, c)$
(4) 0.5: $\forall s_1, s_2, c \quad F(s_1, s_2) \wedge A(s_1, c) \implies A(s_2, c)$	$\neg F(s_1, s_2) \vee \neg A(s_1, c) \vee A(s_2, c)$

and the rest to false. This solution has a cost of 0.5. Important to note that this may not be the only solution with cost 0.5. Each MLN problem can have multiple optimal solutions. Our goal is to return one of them in the fastest manner possible.

Given the previous example each weighted first-order formula can be converted to clausal form and each fact can be represented by a unary hard clause. Table 2.1 explains how each weighted first-logic formula presented in figure 2.1 can be converted to clausal form. It is now possible to instantiate the facts over the relations and generate new clauses in conjunctive normal form (CNF). Note that CNF is used to represent clauses in both SAT and MaxSAT problems. If all the facts are instantiated over the relations, a full grounding of the problem in CNF is generated.

A fully grounded MLN can thus be considered as a weighted MaxSAT problem. An MLN can also be partially grounded if not all the facts are instantiated. Both the fully grounded MLN and the weighted MaxSAT problem have the same goals: first, they can not violate any hard constraints/clauses (soundness); secondly, both try to minimize the sum of the weights of the falsified soft constraints/clauses (selecting the optimal solution).

2.4 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) combines SAT and Theory Solvers. Similarly to SAT solvers, SMT solvers receive an input formula and try to find an assignment for each variable such that the formula is satisfiable. SMT can be seen as a generalization of SAT. The input formula of SMT is a restriction in first-order logic.

Often, there are some applications that require determining the satisfiability of formulas in more expressive logics than CNF [2]. Since the formulas for SAT are presented in CNF, they lack expressiveness. In other words, it is really hard to understand directly what the formula wants to achieve. SMT tries to fix that problem by using first-order logic formulas. This improves expressiveness but loses efficiency.

Given a decidable theory T, a T-atom is a ground atomic formula in theory T. A theory is decidable when all of its models are finite. Given a T-formula, the goal of SMT is to decide if there is one assignment to the variables such that the formula is satisfied. A T-formula is similar to a propositional formula, but it consists of T-literals. A T-literal can be a T-atom or its complement. The domain of the variables depends of the theory T. Their domain can be integer, boolean, real or many other data types.

$x + 2 * y \geq 0$ is a simple example of an SMT formula. $x = -1$ and $y = 0$ is one of the many possible assignments that satisfies this formula.

SMT solvers are used in real life applications for verification, proving the correctness of programs, software testing and many more.

2.5 Overview

In a nutshell, this chapter describes the goals and the dissimilarities of SAT, MaxSAT and MLN. We also briefly introduce SMT and its advantages and disadvantages. We show how can a CNF formula be displayed and the possible gains of using weighted first-order logic formulas. Regarding directly with our algorithm, the main aspect to retain from this chapter is that a fully grounded MLN can be considered as a weighted MaxSAT problem. We use this notion throughout the computation.

Chapter 3

Algorithms for MLN

Due to artificial intelligence, combining logic and probability has been a studied research field for at least a few decades [41]. This chapter overviews some grounding techniques and the progress/evolution of the work related with solving both Maximum Satisfiability (MaxSAT) and Markov Logic Network (MLN) problems. The first MaxSAT algorithms mainly consist of iteratively calling a SAT solver and managing an upper or lower bound, while the more developed MaxSAT algorithms use information obtained in each iteration to guide the search. Finally, the last MaxSAT algorithm in this chapter uses an incremental SAT method. Important to note that this last algorithm is used in ImPROV, that is our algorithm. Thereafter, we overview three MLN algorithms. Note that ImPROV has a similar algorithmic structure to the last algorithm of this chapter, that is the Inference via Proof and Refutation algorithm.

In MLN, one of the differences throughout the years has been the approach of each algorithm in the grounding phase. In this phase, constraints are grounded by instantiating the relations over all facts in their respective domain [26]. This phase can be accomplished by pursuing an eager approach, a lazy approach or even a combination of both. An eager method grounds a set of constraints immediately (without knowing if they are going to be used) while a lazy method only grounds these constraints when they are needed.

Lazy approaches have been crafted to deal with the (large) size of the problem. Real world situations are usually very large. Therefore, most of the recently presented algorithms do not explore all the constraints extensively in the beginning of the method.

3.1 Grounding techniques

During the grounding phase, Tuffy [37] uses a bottom-up approach expressing grounding as a sequence of SQL queries. Using its own algorithm, a set of clauses is selected to be eagerly grounded. This selection can produce scalability issues (if the set is excessively large) or downgrade the quality of the solution (if the set is too small).

On the other hand, lazy inference techniques [40] take advantage of the fact that most grounded facts have a specific value that is generally more common than others. Therefore, these techniques initiate all

Evidence:
$Teaches(D, Compilers)$
$Friends(J, P)$
$Popular(D)$
$\neg Attends(P, Compilers)$

Grounding:
(1) 1.0: $\neg Friends(J, P) \vee Friends(P, J)$
(1) 1.0: $\neg Friends(P, J) \vee Friends(J, P)$
(1) 1.0: $\neg Friends(J, J) \vee Friends(J, J)$
(1) 1.0: $\neg Friends(P, P) \vee Friends(P, P)$
(2) 0.7: $\neg Teaches(D, Compilers) \vee \neg Likes(J, D) \vee Attends(J, Compilers)$
(2) 0.7: $\neg Teaches(D, Compilers) \vee \neg Likes(P, D) \vee Attends(P, Compilers)$
(3) 0.6: $\neg Teaches(D, Compilers) \vee \neg Popular(D) \vee Attends(J, Compilers)$
(3) 0.6: $\neg Teaches(D, Compilers) \vee \neg Popular(D) \vee Attends(P, Compilers)$
(4) 0.5: $\neg Friends(J, P) \vee \neg Attends(J, Compilers) \vee Attends(P, Compilers)$
(4) 0.5: $\neg Friends(P, J) \vee \neg Attends(P, Compilers) \vee Attends(J, Compilers)$
(4) 0.5: $\neg Friends(J, J) \vee \neg Attends(J, Compilers) \vee Attends(J, Compilers)$
(4) 0.5: $\neg Friends(P, P) \vee \neg Attends(P, Compilers) \vee Attends(P, Compilers)$

Figure 3.1: Full grounding of the MLN presented in figure 2.1 (clausal form).

ground facts with a default value and only ground new clauses when these values can be changed for a better goal.

Soft-CEGAR [4] tries to combine both approaches: eagerly grounds the soft constraints and solves the hard constraints in a lazy manner.

Figure 3.1 shows the full grounding of the MLN formula of figure 2.1 in clausal form. Considering that a fully grounded MLN can be regarded as, or converted to a MaxSAT problem, this figure (fig 3.1) is a small example of what could be used in our application.

There is a particular case in the grounding phase of our tool that goes beyond the instantiation of the facts over the relations. The overall MaxSAT formula is a conjunction of all added clauses. Thus, it is not straightforward if we need to add a rule that contains a conjunction. Consider that we want to add ($x_1 \wedge x_2$) to the MaxSAT formula. If ($x_1 \wedge x_2$) is hard, then there is no problem. We add two new hard clauses: (x_1) and (x_2). Both need to be satisfied in order to satisfy the overall formula.

However, if we were to add (x_1) and (x_2) as soft clauses with weight 0.5 then if x_1 and x_2 are false, both soft clauses would not be satisfied and the solution cost would be 1. Note that the original rule ($x_1 \wedge x_2$) has a cost of 0.5 if it is not satisfied.

Consequently, in order to add it to the MaxSAT formula, we create a new auxiliary variable aux_1 and add two hard clauses and one soft clause. The hard clauses are ($x_1 \vee aux_1$) and ($x_2 \vee aux_1$). The soft clause is ($\neg aux_1$, 0.5). If at least one of the variables (x_1 or x_2) takes value false, then the new auxiliary variable aux_1 must take value true so that both hard clauses are satisfied. As a result, the overall formula has a cost of 0.5. However, if x_1 and x_2 both have value true, then aux_1 takes value false and the overall formula has a cost of 0.

Algorithm 3.1: Relax Clauses Function

```
1 Function relaxCls(V_R, φ, ψ)
    Input: V_R, φ, ψ
    Output: updated V_R and φ
2   (R_0, φ_0) ← (V_R, φ)
3   foreach (c, w) ∈ ψ do
4       R_0 ← R_0 ⋃ {r}                      // r is the new relaxation variable
5       c_R ← c ⋃ {r}
6       φ_0 ← φ_0 \ {(c, w)} ⋃ {(c_R, w)}
7   return (R_0, φ_0)
```

3.2 Algorithms for MaxSAT

In order to solve Markov Logic Network instances, some algorithms use MaxSAT extensively. In light of this practice, we present an overview of some MaxSAT algorithms.

In the early MaxSAT solving days, the algorithms used were based on Stochastic Local Search (SLS) [17, 46, 45] and aimed to approximate the MaxSAT solution.

Another used technique consists of converting the MaxSAT problem into a known optimization problem such as an Integer Linear Program (ILP) problem [52], where several approaches take advantage of recent SAT solvers with clause learning and backjumping techniques [12, 48, 35]. Another option is converting the MaxSAT problem to a Maximum Answer Set Programming (MaxASP) problem [15], where recent techniques take advantage of algorithms with a branch and bound approach [38].

A large group of optimal MaxSAT solvers adopt a branch and bound algorithm [9, 16, 24]. On the other hand, in recent years, algorithms that iteratively call a SAT solver have proved to be better than other algorithms, for industrial benchmarks[1].

This group of algorithms iteratively calls a SAT solver until it finds an optimal solution. In order to find the optimal solution, these algorithms maintain a lower and an upper bound on the solution cost. These bounds can be calculated using relaxation variables. A relaxation variable is added to each soft clause such that whenever the soft clause is unsatisfied, the relaxation variable is assigned to true.

Morgado et al. [34] present a relaxation function `RelaxCls`(R, ϕ, ψ). Algorithm 3.1 presents the adapted pseudo-code for the relaxation process. It receives two sets and a formula as arguments. The first argument is a set of relaxation variables, the second argument is a Weighted Conjunctive Normal Form (WCNF) formula and the third argument is a set of clauses. This set must be a subset of ϕ, i.e. $\psi \subseteq \phi$. The procedure relaxes each clause in ψ. Line 4 creates a new relaxation variable and adds it to the set. The newly created relaxation variable is added to the current clause creating a new clause c_R in line 5. Line 6 replaces the old clause with the new relaxed clause. Finally, line 7 returns the updated set of relaxation variables and the updated WCNF formula. This process is used throughout this section and it is called by most of the algorithms presented.

The iterative SAT solver call that is used during these MaxSAT algorithms has as input a WCNF formula. This SAT solver call returns a triple, presented throughout this section as (st, ν, ϕ_C). st has the

[1] http://www.maxsat.udl.cat/

```
Algorithm 3.2: Linear Search Sat-Unsat Algorithm
1  Function LinearSearchSU(φ = φ_H ∪ φ_S)
       Input: φ = φ_H ∪ φ_S
       Output: optimal assignment to φ
2      (st, ν, φ_C) ← SAT(φ_H)                      // check if the set of hard clauses is UNSAT
3      if st = UNSAT then
4          return UNSAT
5      (V_R, μ, ν_sol) ← (∅, Σ_{c_i∈φ_S} w_i, ∅)
6      (V_R, φ_W) ← RelaxCls(V_R, φ, φ_S)                               // relax soft clauses
7      while true do
8          (st, ν, φ_C) ← SAT(φ_W ∪ {CNF(Σ_{r_i∈V_R} w_i · r_i ≤ μ)})      // SAT solver call
9          if st = UNSAT then
10             return ν_sol
11         ν_sol ← ν                                                    // save solution
12         μ ← Σ_{r_i∈V_R ∧ ν(r_i)=true} w_i − 1                        // update bound
```

information whether the formula given as input is SAT or Unsatisfiable (UNSAT). If the formula is SAT, ν has a complete satisfying assignment and ϕ_C is empty ($\emptyset$). Otherwise, if the formula is UNSAT, ν is empty and ϕ_C contains an unsatisfiable subformula $\phi_C \subseteq \phi$ (also known as an UNSAT core). Notice that these UNSAT cores are not used in algorithms 3.2 and 3.3.

Algorithms based on linear search only refine one (either lower or upper) bound during their search (see algorithm 3.2). These algorithms start by relaxing all soft clauses. Then, they try to refine their bound at each iteration.

Algorithms that iteratively improve the upper bound (called Linear Search Sat-Unsat (LSSU) [34]) present the formula as satisfiable for all the SAT solver calls except the last one. SAT4J [3] and QMaxSAT [23] are two of various MaxSAT tools that follow a LSSU strategy.

Algorithm 3.2 contains the pseudo-code for LSSU. Its input is a WCNF formula. In lines 2, 3 and 4 the algorithm does a test to know if the hard clauses have at least one satisfying model. If there is no satisfying model for the set of hard clauses then the algorithm ends and it is returned UNSAT. In line 6, each soft clause is relaxed (using algorithm 3.1). In the final loop, a SAT solver is called iteratively with the new formula obtained (union of hard clauses and relaxed soft clauses) and a newly added constraint that bounds the solution cost. Whenever the SAT solver returns UNSAT, the algorithm returns the solution obtained in the previous iteration. Otherwise, if the SAT solver returns SAT, the current solution is saved (line 11) and the upper bound is updated (line 12).

On the other hand, algorithms that iteratively refine the lower bound (called Linear Search Unsat-Sat (LSUS) [34]) find the working formula to be UNSAT for all the SAT solver calls with the exception of the last call.

Algorithm 3.3 contains the pseudo-code for LSUS. It receives a WCNF formula as input. In lines 2, 3 and 4 the algorithm does a test to know if the hard clauses have at least one satisfying model. If there is no satisfying model for the set of hard clauses then the algorithm returns UNSAT. In line 6, each soft clause is relaxed, that is, a relaxation variable is added to each soft clause. In the final loop, a SAT solver is called iteratively with the new formula obtained (union of hard clauses and relaxed soft clauses) and

Algorithm 3.3: Linear Search Unsat-Sat Algorithm

```
1  Function LinearSearchUS(φ = φ_H ∪ φ_S)
     Input: φ = φ_H ∪ φ_S
     Output: optimal assignment to φ
2    (st, ν, φ_C) ← SAT(φ_H)          // check if the set of hard clauses is UNSAT
3    if st = UNSAT then
4      return UNSAT
5    (V_R, λ, ν) ← (∅, 0, ∅)
6    (V_R, φ_W) ← RelaxCls(V_R, φ, φ_S)                        // relax soft clauses
7    while true do
8      (st, ν, φ_C) ← SAT(φ_W ∪ {CNF(Σ_{r_i ∈ V_R} w_i · r_i ≤ λ)})   // SAT solver call
9      if st = SAT then
10       return ν
11     λ ← UpdateBound({w_i | r_i ∈ V_R}, λ)                    // update bound
```

a newly added constraint that bounds the solution cost. Whenever the SAT solver returns SAT, the algorithm terminates with the respective satisfying model. Otherwise, if the SAT solver returns UNSAT, the lower bound is refined (line 11). This refinement uses the weights of the clauses in the formula to determine what is the next possible value for the lower bound λ. Instead of iterating λ by one in each iteration, this method permits the algorithm to jump several SAT solver calls. Algorithm 3.4 also uses the same refinement at each iteration.

The methods presented thus far (that iteratively call a SAT solver) only use the information about the satisfiability of the formula (whether it is SAT or UNSAT) and the satisfying model (when the formula is SAT).

More recently, algorithms based on SAT solver calls report more information in each call than before. In particular, SAT solvers are able to provide UNSAT cores [53]. An UNSAT core is returned by the SAT solver call whenever there is an UNSAT outcome. These cores can be used to guide the MaxSAT search. These algorithms are called core-guided MaxSAT algorithms [34].

The solving scheme of these algorithms is similar to the ones based on linear search.

Algorithm 3.4 contains the pseudo-code for the MSU3 algorithm presented by Marques-Silva and Planes [29]. It was adapted from the work of Morgado et al. [34]. The input of this algorithm is a WCNF formula. Lines 2, 3 and 4 test if the set of hard clauses alone is satisfiable. If it is not satisfiable, the algorithm ends and returns UNSAT. In line 5 the variables of the algorithm are initialized: the set of relaxed clauses (V_R) is empty, and the lower bound (λ) is 0. In the main loop, a SAT solver is called with the working formula (it changes throughout the algorithm) and a constraint that limits the total weight of the relaxation variables used. If the SAT solver returns SAT, the solution obtained ν is returned. Otherwise, if the solver returns UNSAT, it also returns an UNSAT-core ϕ_C. In this case, the soft clauses that belong to ϕ_C are relaxed (line 10) and the lower bound is refined (line 11). This refinement is the same as the one used in the LSUS algorithm.

This core-guided algorithm (MSU3) is relatable with the LSUS algorithm in many ways. Both start from a lower bound and focus their search in finding the first SAT instance. The main difference between these two algorithms is the phase where the soft clauses are relaxed. In LSUS each soft clause is

Algorithm 3.4: The WMSU3 Algorithm [29]

```
1  Function MSU3 (φ = φ_H ∪ φ_S)
     Input: φ = φ_H ∪ φ_S
     Output: optimal assignment to φ
2    (st, ν, φ_C) ← SAT(φ_H)                  // check if the set of hard clauses is UNSAT
3    if st = UNSAT then
4      return UNSAT
5    (V_R, φ_W, λ) ← (∅, φ, 0)
6    while true do
7      (st, ν, φ_C) ← SAT(φ_W ∪ {CNF(∑_{r_i ∈ V_R} w_i · r_i ≤ λ)})   // SAT solver call
8      if st = SAT then
9        return ν
10     (V_R, φ_W) ← RelaxCls(V_R, φ_W, Soft(φ_C ⋂ φ))              // relax core
11     λ ← UpdateBound({w_i | r_i ∈ V_R}, λ)                        // update bound
```

relaxed in the beginning of the algorithm while MSU3 relaxes the soft clauses on demand, meaning they are only relaxed in the main loop, after each UNSAT-core is returned by each SAT solver call. MSU3 aims to use only one relaxation variable per soft clause, as well as using a small number of total relaxation variables throughout the algorithm.

Fu and Malik [14] introduced a significant and limited core-guided MaxSAT algorithm. It was limited because it was only capable of solving unweighted partial MaxSAT. At each iteration, a new UNSAT core is extracted. A relaxation variable is added to each soft clause that belongs to the UNSAT core and a new constraint limiting the total weight of relaxation variables that are added to the formula.

There were some improved versions of this algorithm [28, 29]. It was also extended in order to solve the weighted partial MaxSAT case [1, 27].

Algorithm 3.5 contains the pseudo-code of *Fu & Malik* for solving weighted partial MaxSAT (WPMS) using SAT incrementally [30]. In the beginning of the algorithm, every soft clause is extended with a fresh blocking variable. In line 2, the working formula ϕ_W is initialized with the set of hard clauses ϕ_H and all the already extended soft clauses. Throughout the algorithm, the assumptions set $\mathcal{A}$ works as an enabler or disabler of soft clauses. Consider, as an example, the clause (x_1). This clause can be extended with a blocking variable b_1. Now that we have the clause $(x_1 \vee b_1)$ we can either enable or disable this clause. If we add b_1 to the assumptions set $\mathcal{A}$ then (x_1) is disabled since $(x_1 \vee b_1)$ is satisfied by b_1. On the other hand, if we add $\neg b_1$ to the assumptions set $\mathcal{A}$, the SAT solver needs to satisfy (x_1) in order to satisfy the overall formula. An assumption controls the value of a variable for a given SAT call, whereas a unit clause controls the value of a variable for all the SAT calls after the unit clause has been added. The key difference between an assumption and an unit clause is that in order to change a unit clause, one needs to destroy the current SAT solver and recreate it from the ground up. The use of a set with assumptions permits the user to flip the value of a variable without needing to destroy the SAT solver. This distinction is most important since the goal of this algorithm is to be incremental and use the same SAT solver throughout the computation. As an initialization, the set $\mathcal{A}$ contains the negation of every blocking variable, thus enabling all soft clauses (line 3). Lines 4-20 contain the main loop of the algorithm.

Algorithm 3.5: Fu-Malik Algorithm with Incremental SAT [30]

```
1  Function Fu-Malik (φ = φ_H ∪ φ_S)
      Input: φ = φ_H ∪ φ_S
      Output: optimal assignment to φ
2     φ_W ← φ_H ∪ {c ∪ {blockingVar(c)} | c ∈ φ_S}          // fresh blocking variables
3     A ← {¬blockingVar(c) | c ∈ φ_S}                       // enable soft clauses
4     while true do
5        (st, ν, φ_C) ← SAT(φ_W, A)                          // SAT solver call
6        if st = SAT then
7           return ν
8        V_R ← ∅
9        m_C = min{w_c | c ∈ φ_C ∧ Soft(c)}
10       foreach c ∈ φ_C ∧ Soft(c) do
11          V_R ← V_R ∪ {r}                                 // r is fresh relaxation variable
12          c_r ← (c \ {blockingVar(c)}) ∪ {r} ∪ {b_r}       // b_r is a fresh variable
13          A ← A ∪ {¬b_r}                                  // enable c_r
14          φ_W ← φ_W ∪ {c_r}
15          w_{c_r} ← m_C
16          if w_c > m_C then
17             w_c ← (w_c − m_C)
18          else
19             A ← (A \ {¬blockingVar(c)}) ∪ {blockingVar(c)}   // disable c
20       φ_W ← φ_W ∪ {CNF(Σ_{r∈V_R} r ≤ 1)}
```

Each iteration starts by having a SAT solver call (line 5). If the working formula ϕ_W is satisfiable, then the algorithm returns the solution found. Note that this solution is optimal. Otherwise, the SAT solver returns an unsatisfiable subformula $\phi_C \subseteq \phi$ (also known as an UNSAT core). For each soft clause c in ϕ_C, the algorithm creates a new clause c_r from c with two extra variables: a relaxation variable r and a fresh blocking variable b_r (line 12). Note that line 13 enables the newly formed clause c_r. In this case, the relaxation variable r represents if the original clause is satisfied (or not) in the MaxSAT solution. Line 9 calculates the weight for the newly formed clauses c_r. It is the minimum weight of all soft clauses in ϕ_C. Soft clauses $c \in \phi_C$ with the same weight as m_C are disabled in line 19. On the other hand, soft clauses $c \in \phi_C$ with weight larger than m_C are not disabled. Their weight is decreased by m_C, thus resulting in a clause split, since the original weight is divided between c and its relaxation c_r.

Important to note that since the working formula is always expanded, the SAT solver is never rebuilt and its internal state is kept (including the learnt clauses). *Fu & Malik* with incremental blocking (algorithm 3.5) significantly outperforms the non-incremental algorithm. Incremental blocking not only solves more instances but is also significantly faster than the non-incremental algorithm.

This section presents the evolution throughout the years of the techniques used to solve MaxSAT problems. These MaxSAT algorithms started by just using bounds and SAT solver calls. Nowadays, the newer and better algorithms still use the same basis: use SAT solver calls and update a bound at the end of each iteration. However, these newer MaxSAT algorithms used the evolution of SAT solvers to develop over the years. The evolution of SAT solvers permits them to return UNSAT cores whenever the SAT solver call is unsatisfiable. These UNSAT cores are used to guide the search in the MaxSAT

algorithms. An important point to consider is that any of these MaxSAT algorithms can be used as a tool when solving the MLN problems. Specifically in our approach, the MaxSAT call in the main loop can use any method from the ones presented in this section. In our case, the MaxSAT solver we use (Open-WBO [31]) uses the last algorithm from this section (algorithm 3.5).

3.3 Algorithms for MLN

In this section, we present the evolution on how to solve Markov Logic Network (MLN) problems throughout the years. Important to note that the algorithms presented in this section solve these problems using a Maximum A Posteriori (MAP) inference method. MAP inference can be used to find the most likely state of world given some evidence. Note that the same problem could be solved using a different inference and therefore have a distinct solution.

Riedel [42] introduced the CPI method. It offered a way to improve existing algorithms such as MaxWalkSAT (MWS) [20] and ILP [52]. It was inspired by the Cutting Plane Method (CPM) [8].

CPI tries to get around the problem of utilizing the full grounding of an MLN by incrementally using only part of the problem instead of the whole network. It optimizes the solution over time and solves these smaller problems using a Maximum A Posteriori (MAP) inference method. Most of the time, these smaller problems are easier to solve and less complex than the entire problem.

Algorithm 3.6 contains the pseudo-code for CPI [42]. Its input is an MLN M, an initial partial grounding G^0, a set of observations x and an integer $maxIterations$. G^i contains a partial grounding for each iteration. The partial grounding for the first iteration (G^0) is made by the formulas that only contain one hidden predicate, making it simple to compute. If, for example, there is a formula $\forall s_1, s_2$ $Friends(s_1, s_2) \implies Friends(s_2, s_1)$ and a fact Friends("J", "P"), that is, a formula that only contains one hidden predicate, it is trivial to compute the fact Friends("P", "J"). In the main loop, line 6 computes the solution of the partial grounding. If the score is higher than the previously saved score then the solution is updated in line 8. At the end of each iteration (line 10) the partial grounding is updated. $Separate(\phi, w, \nu, x)$ grounds the tuples in formulas $(\phi, w) \in M$ that are not yet grounded and that are falsified by the current solution $\nu \cup x$ [43]. The algorithm ends if two consecutive iterations have the same partial grounding or if a predetermined number of iterations is reached (line 11). The maximum number of iterations is used because the algorithm cannot guarantee how many iterations it takes to have a stable solution.

Riedel [42] shows that using CPI in conjunction with a base solver (used in line 6) is better than using the solver alone. Later, Riedel [43] analyses in detail the usage of CPI with different base solvers, such as MWS. It is also formally demonstrated that CPI's accuracy depends on the base solver's accuracy.

More recently, the Soft-Cegar algorithm was introduced [4]. It computes a MAP solution for a given MLN. The Soft-Cegar algorithm was based on three ideas. Counterexample-guided abstraction refinement [6], theorem proving [10] and generalization techniques for accelerating convergence of the loop [32].

This algorithm separates the axioms and the rest of the constraints. The authors consider axioms

Algorithm 3.6: CPI Algorithm

```
1  Function CPI(M, G^0, x, maxIterations)
     Input: M, G^0, x, maxIterations
     Output: solution for M with highest score
2    i ← 0
3    ν_sol ← 0
4    repeat
5      i ← i + 1
6      ν ← solve(G^{i−1}, x)                                   // user's solver
7      if s(ν, x) > s(ν_sol, x) then
8        ν_sol ← ν
9      foreach (φ, w) ∈ M do
10       G^i_φ ← G^{i−1}_φ ∪ Separate(φ, w, ν, x)              // update grounding
11   until G^i_φ = G^{i−1}_φ or i > maxIterations
12   return ν_sol                                               // best found solution
```

as formulas in the MLN that must be satisfied or violated depending on their weight. Soft-Cegar also permits the user to choose the relational MAP solver used throughout the main loop.

Algorithm 3.7 contains the pseudo-code for the Soft-Cegar algorithm [4]. It receives an MLN M as input. Line 2 initializes F_{approx} with all the weighted constraints in M except the axioms in $A(M)$. C is $\emptyset$ in the beginning of the algorithm (line 3). Lines 4-17 describe the main loop of the algorithm. In line 5 an approximation of the input M is created. Then, w_{approx} gets an approximate solution in line 6. Notice that the solver can be an off-the-shelf relational MAP solver, such as Tuffy [37] or Alchemy [22]. The loop in lines 8-14 checks whether the solution (w_{approx}) satisfies the axioms $A(M)$. If any of the axioms are not satisfied by the approximate solution w_{approx}, then they are collected in C'. Next, the algorithm tests if the set C' is $\emptyset$ (line 15). If C' is $\emptyset$, then w_{approx} respects all axioms $A(M)$ and the solution to M is returned (line 16). At the end of each iteration of the main loop, the original algorithm [4] uses a generalization procedure in order to accelerate the convergence of the loop. This procedure is not necessary to have a solution. Nevertheless, it has the potential to reduce the number of iterations needed in the main loop.

It is proved that the Soft-Cegar algorithm returns an exact MAP solution assuming that the MAP solver used during the algorithm (line 6) always returns exact MAP solutions [4]. The authors compare Soft-Cegar with two state-of-the-art relational engines: Alchemy [22] and Tuffy [37] over four real world applications. These applications are Advisor Recommendation [4], Entity Resolution [49], Information Extraction [39] and Relational Classification [37]. This comparison consists of one instance per application. Note that the authors used Tuffy as the underlying relation MAP solver (used in line 6).

In a nutshell, Soft-Cegar is faster and gives better solutions (lower cost) than the other two engines over all four instances. Chaganty et al. [4] explain the results obtained in these experiments in more detail.

The Inference via Proof and Refutation (IPR) algorithm [26] is an iterative algorithm that follows three important pillars: eager proof exploitation, lazy counterexample refutation and termination with soundness and optimality. IPR eagerly analyses the relational constraints in order to produce an initial

Algorithm 3.7: Soft-Cegar Algorithm

```
1  Function Soft-Cegar(M)
       Input: MLN M
       Output: MAP solution
2      F_approx ← F \ A(M)
3      C ← ∅
4      while true do
5          M_approx ← F_approx ∪ C
6          w_approx ← Solve(M_approx)                  // call relational MAP solver
7          C' ← ∅
8          foreach w : ∀¬x.F(¬x) ∈ A(M) do
9              if w = 1.0 then
10                 T ← IsConsistent(w_approx, ¬F)
11                 C' ← C' ∪ {1.0 : f(¬c) | ¬c ∈ T}
12             else
13                 T ← IsConsistent(w_approx, F)
14                 C' ← C' ∪ {0.0 : f(¬c) | ¬c ∈ T}
15         if C' = ∅ then
16             return w_approx                         // return solution found
17         C ← C ∪ C'
```

grounding. This step is used to speed up the convergence of the algorithm. At each iteration, after a solution is found IPR lazily grounds the constraints that are falsified by the current solution. If an exact WPMS solver is used, IPR's termination check assures the soundness and optimality of the solution [26].

Algorithm 3.8 contains the pseudo-code for the IPR algorithm [26]. It receives an MLN M composed of two sets of relational constraints as input, one containing hard constraints (M_H) and another containing soft constraints (M_S). Note that the authors assume that the set of hard constraints M_H is satisfiable. Line 2 computes the initial set of hard constraints: the function `initHard`(M_H) receives the entire set of hard clauses M_H and returns the grounding of a subset of M_H. This function exploits the logical structure of the constraints. Line 3 does the same but with the set of soft constraints M_S. After these two lines, φ contains a set of grounded hard constraints and ψ contains a set of grounded soft constraints. Lines 6-21 contain the main loop of the algorithm. Line 10 checks if there are any hard constraints that are falsified by the previous solution ν_{sol}. If there are any falsified constraints, the algorithm adds their grounding to φ' (line 11). In the same way, ψ' is computed by finding the soft constraints that are violated by ν_{sol} (for each loop in lines 12-14). In line 15 the grounded sets of hard clauses and soft clauses are updated. Then, these updated sets φ and ψ are given to the underlying WPMS solver as inputs and a new solution ν is computed (line 16). The new weight is computed in line 17. The calculated weight is the sum of all the weights of the soft constraints that are satisfied by the solution ν. The terminating condition (line 18) is true if the current weight is equal to the one calculated at the last iteration and if all the hard constraints were satisfied by the last solution (φ' is empty). In lines 20 and 21 the solution ν_{sol} and the weight w are updated so they always have the solution and the weight of the previous iteration.

The authors compared the IPR algorithm [26] with Tuffy [37] and CPI [42, 43] using three different benchmarks with three distinct inputs. Note that the authors used LBX [33] (a minimal correction subset

```
Algorithm 3.8: IPR Algorithm
1  Function IPR(M = M_H ∪ M_S)
       Input: MLN M = M_H ∪ M_S
       Output: Assignment ν_sol
2      φ ← initHard(M_H)                                  // initial grounding
3      ψ ← initSoft(M_S)                                  // initial grounding
4      ν_sol ← ∅
5      w ← 0
6      while true do
7          φ' ← ∅
8          ψ' ← ∅
9          foreach h ∈ M_H do
10             if Violation(h, ν_sol) then
11                 φ' ← φ' ∪ ground(h)
12         foreach (s, w) ∈ M_S do
13             if Violation(s, ν_sol) then
14                 ψ' ← ψ' ∪ ground(s)
15         (φ, ψ) ← (φ ∧ φ', ψ ∧ ψ')                     // update hard/soft clauses
16         ν ← WPMS(φ, ψ)                                 // call WPMS solver
17         w' ← Weight(ν, ψ)
18         if w' = w ∧ φ' = true then
19             return ν_sol
20         ν_sol ← ν
21         w ← w'
```

(MCS) extraction algorithm) as the underlying solver (used in line 16). In these nine experiments, IPR concludes every one of them, while CPI concludes eight and Tuffy is only able to terminate five. In short, IPR outperforms the other two approaches in terms of runtime in these instances but has very similar solution costs when compared to CPI. Tuffy presented significantly higher solution costs than the other two.

In summary, this chapter presents all the essential work related to our algorithm that has been developed through the recent years.

Firstly, it describes one of the main phases of all Markov Logic Network (MLN) algorithms: grounding. This grounding can either be eager, lazy or a combination of both. Eager approaches may have scalability problems, while lazy approaches may be slow if it is necessary to ground large numbers of clauses. Hence, most recently proposed algorithms do a combination of both approaches, including IPR and Inference using PROpagation and Validation (ImPROV), our algorithm.

Secondly, we present some of the most known Maximum Satisfiability (MaxSAT) algorithms and their evolution throughout the years. All in all, MaxSAT solvers are the most researched and developed solvers regarding problems that have certain objectives we want to optimize in addition to some rules we do not want to break. Important to note that ImPROV uses the last MaxSAT algorithm presented (algorithm 3.5) in this chapter.

Finally, the end of the chapter describes three algorithms that solve MLN problems, the type of problems our algorithm (ImPROV) solves. All of the MLN algorithms presented have identical algorithmic

structure, that is, an initial phase that usually decreases the algorithm convergence time, followed by a main loop that calls a solver and then proceeds to validate each constraint with the previously computed solution. This structure is also the basis for ImPROV. Our algorithm has the same algorithmic structure as IPR. However, we intend to use an SMT solver to validate all the constraints and use an incremental core-guided MaxSAT algorithm instead of a non-incremental one that must begin the search from the beginning in each iteration. Our main goal is to present a solver that is correct, i.e. a sound and optimal solver. Sound, meaning the solution does not violate any hard constraints and optimal, meaning the lowest possible solution cost. After ensuring the correctness of the solver, our objective is having a reasonable performance that can compete with the state-of-the-art engines.

Chapter 4

ImPROV (Inference using PROpagation and Validation)

In this chapter we describe our software step by step, explaining the progression of the work, the difficulties we had to face and the decisions we took to solve these problems. Firstly, we present the architecture of the tool, its pseudo-code as well as a comparison with previously presented algorithms. Secondly, the initializations, the algorithms and the methods used throughout the computation are identified and explained. Throughout the chapter, we also include some examples.

4.1 Architecture

We assume that the Markov Logic Network (MLN) problem is represented in three files: the MLN file where both the predicates and the rules are described; the evidence file contains all the literals used as facts and the query file informs the algorithm of what predicates it should output in the final solution. The three files are represented in the left column of figure 4.1.

These parsed predicates indicate their name, if they have a closed world assumption, how many arguments they have, as well as the type of each argument. For example, the predicate *Friends(person, person) has the name Friends, it has a closed world assumption ('*' before the name), it has two arguments, the first one being of type person and the second one being of type person as well. A predicate with a closed world assumption forces all ground atoms of this predicate not listed in the evidence to be false. This predicate represents the friendship between two people.

Each parsed rule is either soft or hard. If a weight is present at the beginning then it is a soft rule. Otherwise, if the rule finishes with a period then the rule is defined as hard. If there are both a weight and a period the algorithm processes the rule as hard. These rules are presented in first-order logic form.

Figure 4.2 shows an example of an MLN input file. In this case, the first three uncommented lines represent the predicates Friends, Smokes and Cancer. The lines that start with a real number are the rules of the problem. These constraints are soft because the weight is present in the beginning of the

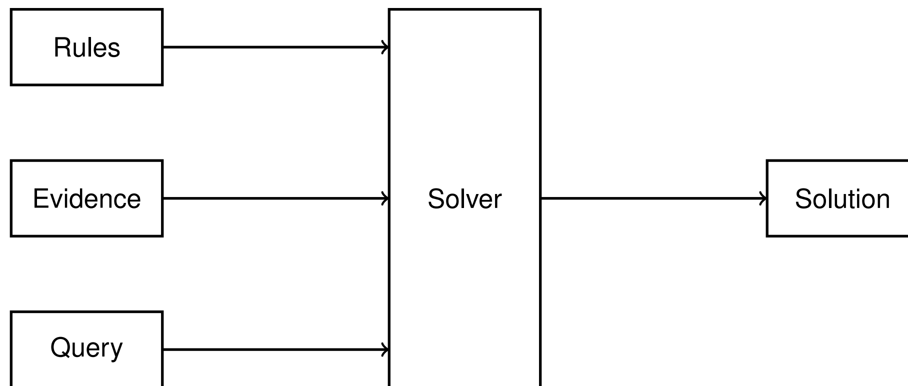


Figure 4.1: Software architecture.

```
// Predicates:
*Friends(person, person)
Smokes(person)
Cancer(person)

// Rules:
0.5: Smokes(a1) ⟹ Cancer(a1)
0.4: Friends(a1,a2) ∧ Smokes(a1) ⟹ Smokes(a2)
0.4: !Friends(a1,a2) ∨ !Smokes(a2) ∨ Smokes(a1)
```

Figure 4.2: Example of an MLN program.

line and there is no period at the end of the line. These rules can be represented in first-order logic. Implications or even equivalences can be used in these rules. Due to the lack of time to complete the software as it was intended, existential quantifiers are not accepted by our tool.

Figure 4.3 contains an example of an evidence input file. It is a list of unary literals, containing a single fact in each line. These facts can be positive or negative. Negative facts contain an exclamation point before the literal. For example, in this case, the sixth line explicitly states that Gary and Frank are not friends. All of these facts need to be satisfied by the solution.

Figure 4.3 contains an example of a query input file. It is an enumeration of the predicates that are shown in the output. This file does not interfere with the computation or the solution. It points out how the output of the solution should be displayed.

Figure 4.4 shows the solution of the problem presented as an example throughout this chapter for the MLN program, evidence and query represented in figures 4.2 and 4.3. There are two important aspects to note: firstly, due to the query, only the Cancer predicate is in the solution; secondly, there are no negative literals in the solution. Our tool only outputs the literals that are positive in the solution. All other literals are implicitly negative.

4.2 Algorithms

These following sections describe all algorithms we use in our tool. The algorithms and approaches presented correspond to the solver in the middle column of figure 4.1. It is in this step that the entire computation is done, from the parsing of the input files to the computation of a solution.

// Evidence:	// Query:
Friends(Anna, Bob) Friends(Anna, Edward) Friends(Anna, Frank) Friends(Edward, Frank) Friends(Gary, Helen) !Friends(Gary, Frank) Smokes(Anna) Smokes(Edward)	Cancer(x)

Figure 4.3: Example of an evidence file and a query file.

```
Cancer(Anna)
Cancer(Edward)
Cancer(Frank)
Cancer(Bob)
```

Figure 4.4: Solution for MLN defined in figures 4.2 and 4.3.

4.2.1 Algorithmic structure

Algorithm 4.1 contains the pseudo-code for the ImPROV algorithm. It receives an MLN M composed of two sets of relational constraints as input, one containing hard constraints (M_H) and another containing soft constraints (M_S). We assume that, throughout the algorithm, the set of grounded hard clauses is always satisfiable. Note that, we could add a SAT solver call only with the grounded hard clauses before each MaxSAT solver call. We tried to incorporate this SAT solver call to our algorithm, however, the MaxSAT solver we use does not support this specific call for each iteration. Since most of the MLN problems contain a satisfiable set of hard constraints, we felt that building a new SAT solver in the beginning of each iteration was unnecessary.

Lines 2 to 10 contain the preprocessing and initializations of the algorithm. The main loop of the algorithm is represented from line 11 to line 29. ImPROV has a similar algorithmic structure as the Inference via Proof and Refutation (IPR) algorithm, presented in section 3.3. ImPROV exploits the structure of Horn constraints and does the same initial propagation as IPR. Throughout the main loop, both algorithms call a Weighted Partial Maximum Satisfiability (WPMS) solver and validate each rule using the current MaxSAT solution. Both main loops are an adaptation of the Linear Search Unsat-Sat algorithm presented as algorithm 3.3. However, there is a key difference between both algorithms: whenever a rule is not validated, the IPR algorithm finds all the counterexamples of the tested rule and grounds them while ImPROV only finds two counterexample per validation call. Note that, if ImPROV (algorithm 4.1) needs k counterexamples, it repeats lines 22-26 k times. The same can be applied when the tested rule is a hard constraint. The termination check of ImPROV (line 28) verifies if all the rules that are not grounded are satisfied by the current MaxSAT solution. As soon as none of the rules (not yet grounded) are falsified by the current MaxSAT solution, the algorithm terminates (line 29). On the other hand, since IPR returns the solution found in the previous iteration, the algorithm needs to check if the current solution cost is equal to the solution cost calculated in the previous iteration. The solution found by both algorithms is optimal, meaning it has the lowest solution cost possible.

Algorithm 4.1: ImPROV Algorithm

```
1  Function ImPROV(M = M_H ∪ M_S, evidence)
     Input: MLN M = M_H ∪ M_S, evidence
     Output: Assignment ν_sol
2    φ ← evidence                                   // evidence from input
3    (ψ, φ', ψ') ← (∅, ∅, ∅)
4    foreach r ∈ M do
5      convertRule(r)                               // Convert from first-order logic to CNF
6      if isHorn(r) then
7        (φ', ψ') ← initialPropagation(r, evidence) // eager initial propagation
8        (φ, ψ) ← (φ ∧ φ', ψ ∧ ψ')                  // update hard/soft clauses
9    foreach (s_i, w_i) ∈ M_S do
10     CE_i ← ∅                                     // initialization of counterexample set
11   while true do
12     φ' ← ∅
13     ψ' ← ∅
14     rulesValidated ← true
15     ν_sol ← WPMS(φ, ψ)                           // call WPMS solver
16     foreach h ∈ M_H do
17       (st, ν_SMT) ← SMT(¬h ∧ ν_sol)              // SMT call to validate hard constraint
18       if st = SAT then
19         φ' ← φ' ∪ ν_SMT                          // add SMT's counterexample to the current grounding
20         rulesValidated ← false
21     foreach (s_i, w_i) ∈ M_S do
22       (st, ν_SMT) ← SMT(¬(s_i) ∧ ν_sol ∧ CE_i)   // SMT call to validate soft constraint
23       if st = SAT then
24         ψ' ← ψ' ∪ ν_SMT                          // add SMT's counterexample to the current grounding
25         CE_i = CE_i ∪ ¬ν_SMT                     // add negation of counterexample to set
26         rulesValidated ← false
27     (φ, ψ) ← (φ ∧ φ', ψ ∧ ψ')                    // update hard/soft clauses
28     if rulesValidated = true then
29       return ν_sol
```

4.2.2 Preprocessing and Initializations

This section describes ImPROV's lines 2-8 in detail. These lines represent the preprocessing of the algorithm. Since MaxSAT solvers can not deal with first-order logic formulas, in the beginning of the algorithm, these formulas or constraints have to be converted to Conjunctive Normal Form (CNF) (line 5), so that the MaxSAT solver can solve the formulas presented in each iteration. Note that throughout the algorithm, we maintain a bidirectional map in order to preserve the information about both the literals from the MLN and the MaxSAT variables.

Before solving the MLN instance we need to construct our Satisfiability Modulo Theories (SMT) solver. The SMT solver can be used in default mode or it can be used with a tactic or even a combination of tactics. Tactics are the basic building block for creating custom solvers for specific problem domains. In other words, the use of tactics makes the SMT solver follow certain heuristics that may be highly tuned for a specific problem. In our algorithm, we use the Z3 SMT solver [10]. In Z3, it is possible to create custom strategies using the available basic building blocks. However, in our case, we use a built-in tactic

called macro-finder. This macro-finder tactic expands quantifiers representing macros at preprocessing time. Since we use several universal quantifiers in our SMT formula for each iteration, the use of this tactic significantly improves the search and therefore saves some time per SMT solver call.

After these initializations are done, our algorithm can be divided into two distinct parts. In order to speed up the search and diminish the number of iterations, our tool uses an initial eager propagation. Subsequently, it uses a lazy method in the main loop of the algorithm.

In order to do our initial eager propagation (lines 6-8) we check if there are any Horn constraints in the input. A Horn constraint [18] is a disjunction of literals with at most one positive literal. It is typically written in the form of an inference rule ($u_1 \wedge ... \wedge u_n \implies u$). The three rules presented in figure 4.2 are examples of Horn constraints. The first and the second one are presented in the form of an inference rule and the third is presented in CNF. Our eager exploration is done only with Horn constraints. If these Horn constraints contain only one single hidden predicate, our algorithm propagates the current values and new literals are discovered. For instance, when our algorithm uses this eager propagation in the running example (figures 4.2 and 4.3), it can find the grounding literal Cancer(Anna) using the first rule. This eager propagation works because the first constraint is a Horn constraint and because there is only one single hidden predicate (in this case Cancer). In other words, since Smokes(Anna) is a fact, Cancer is the only predicate in the first rule that is hidden so the algorithm propagates the values that are currently assigned (Anna) and finds the new grounding literal Cancer(Anna). Notice that this enables the solver to infer new literals without performing a full ground of the rule. We have this eager approach because these instantiated constraints would be found by the lazy part of the algorithm anyhow. If we do them all at once, not only do we save time, but also diminish the number of iterations of the main loop. If there are no Horn constraints or none of the Horn constraints have a single hidden predicate this step is ignored and the algorithm proceeds to the lazy part. Important to note that in practice, most constraints in many inference tasks are Horn [26], which makes this eager exploration useful in most cases. As a result, it tends to improve our algorithm.

4.2.3 Rule validation

This section describes ImPROV's lines 16-27 in detail. These lines represent the validation of each rule. In ImPROV's main loop, after the MaxSAT solver call, our algorithm checks, one by one, if there are any rules that are falsified by the current MaxSAT solution. To do that, we build a formula with the conjunction of the current MaxSAT solution and the negation of the rule we want to validate (lines 17 and 22). Next, the SMT solver is applied to the formula. Section c) of figure 4.5 and section b) of figure 4.6 show the encoding of a negated rule. In this case, it represents the validation of the third rule of figure 4.2.

If the SMT solver returns UNSAT then the rule is not falsified by the current MaxSAT solution. On the other hand, if the SMT solver returns SAT then there is at least one counterexample that falsifies the current MaxSAT solution. The counterexample given by the SMT solver is then grounded and added to the MaxSAT solver (lines 19 and 24). In general, this is how the SMT solver call works. However, there are some subtleties when constructing the SMT formula, particularly the encoding of the MaxSAT

```
a)  (assert (= (Friends 0 1) true))
    (assert (= (Friends 0 2) true))
    (assert (= (Friends 0 3) true))
    (assert (= (Friends 2 3) true))
    (assert (= (Friends 4 5) true))
    (assert (= (Friends 4 3) false))
    (assert (= (Smokes 0) true))
    (assert (= (Smokes 2) true))

b)  (assert (= (Friends x0 x1)
      (or (and (= x0 0) (= x1 1))
          (and (= x0 0) (= x1 2))
          (and (= x0 0) (= x1 3))
          (and (= x0 2) (= x1 3))
          (and (= x0 4) (= x1 5))
          (and (= x0 4) (= x1 3)))))

c)  (assert (not (or (not (Friends x0 x1)) (not (Smokes x1)) (Smokes x0))))
```

Figure 4.5: SMT encoding when validating the third rule. (first method)

```
a)  (assert (forall ((x0 Int) (x1 Int))
      (= (Friends x0 x1)
          (or (and (= x0 0) (= x1 1))
              (and (= x0 0) (= x1 2))
              (and (= x0 0) (= x1 3))
              (and (= x0 2) (= x1 3))
              (and (= x0 4) (= x1 5))))))
    (assert (forall ((x2 Int)) (= (Smokes x2) (or (= x2 0) (= x2 2)))))

b)  (assert (not (or (not (Friends x0 x1)) (not (Smokes x1)) (Smokes x0))))
```

Figure 4.6: SMT encoding when validating the third rule. (second method)

solution.

We tried to use the capabilities of Z3 [10] and tested a different way to configure the current MaxSAT solution into the SMT formula. Z3 allows the use of assumptions. Assumptions are auxiliary propositional variables that we want to track. Note that, assumptions are not really soft constraints, but they can be used to implement them. In our case, we tried to implement the use of assumptions to take the current MaxSAT solution out of the SMT formula. In other words, we tried to use the set of assumptions to hold the boolean variables that represent the current MaxSAT solution. This assumptions set works as an input in the SMT solver, thus making the SMT formula smaller and with less information. The results from using assumptions did not show any improvements, it even drastically slowed some of the instances tested. Considering these results, ImPROV does not use the assumptions feature of Z3.

ImPROV has two different approaches to encode the current MaxSAT solution into the SMT solver.[1]

Our first approach has two steps: firstly, it adds an unary assertion per MaxSAT solution variable. Section a) of figure 4.5 shows how this step is encoded. Note that this encoding refers to the first time the SMT solver is called when solving the example presented in this chapter (input files of figures 4.2 and 4.3). Comparing this encoding with the evidence from figure 4.3 we can infer that Anna is represented by 0, Bob is represented by 1 and so on. These are equivalences that inform the solver if a

[1] The figures that present SMT formulas do not consider the initial eager propagation so that a direct comparison with the input can be explained more easily.

MaxSAT solution variable is true or false.

Section b) of figure 4.5 is needed due to the closed world assumption concept. As explained previously, if a predicate has a closed world assumption then all ground atoms of this predicate that are not listed in the evidence are forced to be false. When encoding the validation of a rule, for each predicate in that rule that has a closed world assumption, this first approach adds an assertion. If the predicate is true, the assertion obliges the SMT solver to choose one instance of the positive evidence. In other words, if the predicate is considered true, the solver can only choose instances of the positive evidence to represent it. Important to note that this second step is only done if there is a predicate that has a closed world assumption in the rule that is currently being validated.

There is an alternative approach that is much more simpler and faster than the first one. It is not dependent on predicates that have a closed world assumption so it has always the same structure regardless the predicate we are encoding. Section a) of figure 4.6 shows how this step can be encoded. Note that this encoding refers to the first time the SMT solver is called when solving the example presented in this chapter (input files of figures 4.2 and 4.3). For each predicate in the current rule that is being validated we encode a ”forall” assertion that makes the SMT solver choose a positive instance of this predicate. In other words, our encoding assumes that every instance of the predicate that is not defined as positive, has a negative value. This method follows the approach of some of the SAT solvers. When these solvers are testing values for variables, they usually test the value false first and then, if the false value does not work, they flip the value of the variable to true.

We use the second approach because it is faster in every example we tested. Not only is it faster than the first approach when using a default SMT solver but it also gains more advantages from using the built-in tactic macro-finder. The second method is faster due to two main aspects: first, this encoding does not need to take into account if the predicates have a closed world assumption, making the formula much simpler; secondly, the encoding of the current MaxSAT solution uses ”forall” assertions that are greatly improved by the built-in tactic macro-finder.

In order to register the counterexamples found we maintain a set CE_i (initialized in line 10) for each soft constraint. Once the SMT solver finds an answer, the negation of this counterexample is added to the respective set so that the solver never finds the same counterexample twice. Note that this step is necessary since the MaxSAT solver does not need to satisfy every soft clause. If, for instance, the counterexample is from a soft rule, the solution from the next iteration of the MaxSAT solver might not satisfy this newly added clause. If so, in the next iteration, the SMT solver might return the same counterexample as it did in the previous iteration. Important to note that the algorithm only needed to add the negation of the counterexample to the the set CE_i if the falsified rule is soft. If the counterexample found is from a hard constraint then, in the next iteration, the variables in the MaxSAT solver are automatically assigned so that the instance of the rule is satisfied, rendering the counterexample just found impossible to be returned as the SMT solution. Note that line 22 calls the SMT solver with the previously explained formula in conjunction with the set of counterexamples already found.

4.2.4 Incremental MaxSAT

This section describes ImPROV's line 15 in detail. This line represents the MaxSAT solver call. This call is the first computation in ImPROV's main loop. When the eager propagation is done, our tool starts the lazy and incremental part of the algorithm. This part of the algorithm is called lazy because from this point on the tool only grounds constraints when it is necessary. It does not ground constraints preemptively. Our main goal with this approach is to minimize the number of constraints and literals that are grounded throughout the algorithm. Our objective in each iteration is to find out which rules are falsified by the current MaxSAT solution, that is, find out which constraints need to be grounded. The constraints and the literals that need to be grounded are then added to the MaxSAT solver. Hence, when beginning a new iteration, the MaxSAT formula has a set of brand new variables and/or clauses due to the grounding done in the last iteration.

On the other hand, this part of the algorithm is also mentioned as incremental since our method only adds variables and clauses to the MaxSAT solver. The algorithm never removes variables or clauses from the MaxSAT solver. Thus, the MaxSAT formula grows with each iteration. Our goal with this approach is to keep expanding the MaxSAT formula without drastically changing its structure. This concept is used to make sure we take full advantage of using an incremental MaxSAT solver. Important to note that any off-the-shelf MaxSAT solver could be used in this part of the algorithm. However, using an incremental MaxSAT solver certifies that the computation of the solver is not restarted in each iteration. It continues directly from the last iteration.

In the beginning of the main loop, there is a MaxSAT solver call with all the current grounded variables and clauses. In our algorithm we choose to use an incremental MaxSAT solver that suits our needs. Since our goal is to lazily ground the problem at hand, the use of a solver that allows the addition of new variables and clauses step by step is ideal in this method. This recently presented MaxSAT solver [47] uses UNSAT cores to guide the search and aims to improve the efficiency on solving a sequence of similar problem instances. It is known as an incremental core-guided MaxSAT algorithm. This algorithm uses the *Fu & Malik* algorithm with Incremental SAT [30] to solve its MaxSAT formulas. The *Fu & Malik* algorithm with Incremental SAT is thoroughly explained in section 3.2. Algorithm 3.5 contains its pseudo-code.

Si et al. [47] consider the case when the sequence is constructed by adding new hard or soft clauses. Now this circumstance reflects our own problem perfectly. The solver's approach is similar to the idea of incremental SAT solving presented by Fu and Malik [14] however there are some significant differences. Consider a sequence of MaxSAT formulas ϕ^1, ϕ^2, ..., ϕ^n, with $\phi^k \subseteq \phi^{k+1}$ and $\phi^k = \phi^k_H \cup \phi^k_S$, where ϕ^k_H contains the hard clauses from ϕ^k and ϕ^k_S contains the soft clauses from ϕ^k. The incremental MaxSAT solver applies the incremental SAT solving algorithm (algorithm 3.5) to ϕ^1 and then extends the resulting working formula ϕ_W with hard clauses from $\phi^2_H \setminus \phi^1_H$ and soft clauses from $\phi^2_S \setminus \phi^1_S$, each extended with a new and fresh relaxation variable. Then resume the main loop of algorithm 3.5 (from line 4). This procedure is repeated for the remainder of the algorithm. In other words, each time algorithm 3.5 returns, i.e. the working formula $\phi_W$is satisfiable, the hard clauses $\phi^{k+1}_H \setminus \phi^k_H$ and the soft clauses $\phi^{k+1}_S \setminus \phi^k_S$ with a fresh blocking variable are added to the working formula ϕ_W. The assumption set $\mathcal{A}$ is also updated.

Subsequently, the algorithm goes to line 4. Note that, since the algorithm only adds clauses, the search does not need to restart from the beginning for each formula in the sequence. The computation and the information learnt can be used for future iterations. Furthermore, the incremental MaxSAT solver does not redo the computation from previous iterations.

This approach is then, incremental at two levels: first, it uses incremental SAT for each MaxSAT formula; second, it uses incremental MaxSAT across the sequence of instances ϕ^1, ϕ^2, ..., ϕ^n.

This approach is correct because the addition of new soft or hard clauses does not invalidate the previously found UNSAT cores [14]. That is to say, if, for instance, there is a set of clauses that makes a certain formula UNSAT, the extension of that said formula with either hard or soft clauses does not make the formula satisfiable in any way. Consider the unsatisfiable CNF formula $(x_1) \wedge (x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_3)$. The UNSAT core returned is $(x_1) \wedge (x_2) \wedge (\neg x_1 \vee \neg x_2)$. Regardless of the addition of clauses such as $(x_4 \vee x_5)$ or $(x_2 \vee x_3)$, the previously found UNSAT core is always valid.

There is a slight obstacle when we try to solve sequential MaxSAT using this incremental core-guided algorithm. If we consider a sequence of MaxSAT instances ϕ^1, ϕ^2, ..., ϕ^n where $\phi^i \subseteq \phi^j$ for $1 \leq i < j \leq n$, the set of UNSAT cores in ϕ^i is a subset of the UNSAT cores in ϕ^j. Therefore, this incremental version might find a poorer UNSAT core than the non-incremental version. Finding a core of poor quality is often detrimental to the rest of the computation.[47]

The authors present a solution that consists on using restarts to the computation. The use of restarts has been used in this area for some time now and it has proved that despite not being a sophisticated technique, it can have crucial impact on performance [44]. The explanation used in previously presented algorithms is that restarts help the solver avoid spending time in branches or sections that would not help the solver to proceed with the search in a timely manner. This incremental core-guided MaxSAT algorithm uses restarts in this exact manner: once the authors suspect that the algorithm is finding UNSAT cores of poor quality, they restart the whole computation. In this case, the poor UNSAT cores found in the incremental core-guided algorithm could be compared with the branches in which there is neither an easy-to-find satisfying assignment nor opportunities for fast learning of strong clauses in SAT solvers.

Si et al. [47] use the amount of clause splitting to ascertain the quality of the UNSAT cores found. As previously mentioned in section 3.2, the *Fu & Malik* algorithm with Incremental SAT (algorithm 3.5) may split some soft clauses during the main loop. A given soft clause $c \in \phi_C$ is split if its weight is larger than the weight of the unsatisfiable core (m_C). This incremental MaxSAT solver maintains a split counter for each soft clause and a split limit. If this limit is hit, the solver is rebuilt and algorithm 3.5 is restarted. In order to maintain completeness, the solver restarts at most once for each MaxSAT formula ϕ^i in the sequential MaxSAT instance.

This new incremental core-guided algorithm was tested in three applications: abstraction refinement [54], user-guided analysis [25] and statistical relational inference [26]. The algorithm is implemented in the Open-WBO [31] MaxSAT solver. Mangal et al. [26] compare the original Open-WBO solver and the incremental-without-restarts algorithms and the original Open-WBO solver and the incremental-with-restarts algorithms. First, the results show that the incremental-without-restarts algorithm performs

significantly worse than the original Open-WBO solver algorithm. These results justify the need for the restart technique. On the other hand, the results from the incremental-with-restarts algorithm show a significant improvement of the overall performance when solving similar MaxSAT instances in the same sequence. The authors specify that using five as the split limit yields the best results out of all the possibilities tested.

Thus, reusing the computation with the incremental-with-restarts algorithm is the perfect fit for our own algorithm. The sequence of MaxSAT instances in ImPROV is similar and can use the full potential of this incremental method.

In a nutshell, this chapter explains ImPROV's development throughout time. First, we present its architecture, the input necessary and a small example that clarifies the process of the algorithm. Then, we introduce its algorithmic structure and the similarities to IPR. Both algorithms are very similar in structure: initial eager propagation followed by lazy section of the algorithm. The rest of the chapter describes each of the main aspects of the algorithm in detail: all the preprocessing and initializations from ImPROV, how we use an SMT solver to validate all the constraints after each MaxSAT solver call and the use of an incremental MaxSAT method.

Chapter 5

Evaluation

In this chapter, we test, analyse and present the results of our tool: Inference using PROpagation and Validation (ImPROV). The chapter is divided into three sections: first, we describe the instances and problems we use to evaluate ImPROV; secondly, we test it using all the possible configurations, so that we can use the best configuration of our tool when we analyse it against the other state-of-the-art tools; lastly, we run these instances with the state-of-the-art algorithms/engines and compare/analyse their results with the ones from ImPROV. All experiments were done on a Linux machine with a 1.4 GHz processor and 132GB of RAM. We set a time out of 30 minutes (1800 seconds) for all the experiments. Throughout the chapter, we use K to represent thousands and M to represent millions.

5.1 MLN Instances

We evaluate ImPROV with problems generated from three different applications: entity resolution (ER), relational classification (RC) and advisor recommendation (AR). Singla and Domingos [49] used Markov Logic Networks to solve instances of the entity resolution problem. The entity resolution problem was introduced by Newcombe et al. [36] and a statistical formulation was proposed by Fellegi and Sunter [13]. It is the problem of finding out which records in a database refer to the same object. It is considered an important step in the data mining process. Relational classification [37] consists of labelling or categorizing papers using some information about them. In this case, we use the author and the references of each paper to give them a proper category. Finally, Advisor recommendation [4] tries to find the ideal PhD advisor for a starting graduate student.

Throughout the evaluation, we use three different instances of each application, with a different number of facts. Table 5.1 describes each instance of each application. The problems from the ER application are presented as ER1, ER2 and ER3 and have 208K grounded clauses in full grounding, on average. The instances from the RC application are the biggest ones when comparing the number of grounded clauses in full grounding, since they have 139M on average. The RC problems are presented as RC1, RC2 and RC3. Lastly, AR1, AR2 and AR3 represent the instances from the AR application and have the least amount of grounded clauses in full grounding: 6K on average.

Table 5.1: Statistics of applications' constraints and databases. (K = thousand, M = million)

	databases	# relations	# rules	# facts	# clauses in full grounding
Entity Resolution	ER1	4	4 hard + 4 soft	219	120K
	ER2			234	190K
	ER3			249	314K
Relational Classification	RC1	3	1 hard + 2 soft	557	26M
	RC2			1006	104M
	RC3			1612	286M
Advisor Recommendation	AR1	3	0 hard + 2 soft	40	3K
	AR2			60	7K
	AR3			68	9K

5.2 ImPROV's configuration

Following the selection of the instances, we set the proper and best configuration for our tool. This configuration is key to the performance of the algorithm. Some different configurations can change the total time of some instances by several minutes.

5.2.1 Initial eager propagation

The first configuration we can set is the initial eager propagation that was thoroughly explained in section 4.2.2. Note that this step is optional and does not change the result. We use it since it helps the algorithm to converge faster, that is, to take less iterations of the main loop to finish the computation.

Table 5.2 indicates how each instance changes when we run the algorithm with or without the initial eager propagation. This table presents the number of iterations of the main loop that are necessary for the computation to finish, as well as the total time the algorithm takes to conclude. As mentioned in section 4.2.2, the initial eager propagation can be used when there are Horn constraints [18] and these Horn constraints only have one hidden predicate.

Table 5.2 shows that three of the instances have different results when using initial eager propagation (RC1, RC2 and RC3), while the remaining six (ER1, ER2, ER3, AR1, AR2 and AR3) do not have any changes (iterations or total time) when using initial eager propagation. These three applications fit this test perfectly since they show the three possible situations.

The instances from the ER application, do not present any changes due to the lack of positive evidence from certain relations. These instances have some Horn constraints, but no positive evidence. The absence of positive evidence of the relations that take part in the Horn constraints cause a high number of hidden predicates. Consider the Horn constraint (¬ SameAuthor(a1,a2) ∨ ¬ SameAuthor(a2,a3) ∨ SameAuthor(a1,a3)). If there is no positive evidence of the relation SameAuthor(person, person), then there are at least two hidden predicates.

The instances from the RC application are the only ones that have a different number of iterations and time in all three cases. When we use the initial eager propagation, the number of iterations of the

Table 5.2: Results of ImPROV with and without the initial eager propagation phase. The instances used are the ones presented in table 5.1. (m = minutes, s = seconds)

	databases	without eager propagation		with eager propagation	
		# iterations	total time	# iterations	total time
Entity Resolution	ER1	465	32s	465	32s
	ER2	1029	3m14s	1029	3m14s
	ER3	1820	14m13s	1820	14m13s
Relational Classification	RC1	253	1m10s	203	1m05s
	RC2	264	4m22s	223	3m48s
	RC3	654	12m54s	494	10m29s
Advisor Recommendation	AR1	1445	1m32s	1445	1m32s
	AR2	3290	12m15s	3290	12m15s
	AR3	4274	25m09s	4274	25m09s

main loop decreases, as well as the total time of the algorithm. This step helps the convergence of the algorithm since it discovers some of the literals upfront. The algorithm finds these literals whether we use the initial eager propagation or not. The difference is in what part of the algorithm do we find these literals. If we use the eager propagation, the algorithm finds these new literals before the main loop (first part of the algorithm). Otherwise, the algorithm finds the literals during the main loop, using the counterexamples from the SMT solver.

The instances from the AR application, do not present any changes due to the lack of Horn constraints. In theses instances there are no Horn constraints so our algorithm can not perform the initial eager propagation.

These nine instances demonstrate the three possible cases when using the initial eager propagation: lack of positive evidence (ER1, ER2 and ER3); use of Horn constraints and positive evidence (RC1, RC2 and RC3); lack of Horn constraints (AR1, AR2 and AR3).

We conclude that the use of this initial eager propagation can not be always used, but when it can, it significantly improves the time of the algorithm by diminishing the number of iterations needed. Note that we used plenty of instances to test this step as it is an important part of our algorithm. Considering the amount of tests on configuration that we present in this chapter, we do not show those results in this section.

Since the use of the initial eager propagation provides better results, from this point forward, all the experiments we perform use the initial eager propagation method.

5.2.2 Counterexample generation

In the second configuration we experiment the number of counterexamples that the SMT (Satisfiability Modulo Theories) solver should return in each validation call. One of the core parts of ImPROV is the rule validation explained in section 4.2.3. In the main loop, after each MaxSAT solver call, ImPROV checks if there is any rule that is falsified by the current MaxSAT solution. In the simplest version of ImPROV, if at least one of the rules is falsified by the current MaxSAT solution then the algorithm calls

the SMT solver and it returns one counterexample.

Our goal with these following experiments is to verify what is the best possible amount of counterexamples that the algorithm should return per validation call. Note that it is not possible to ask for a specific number of counterexamples in Z3 [10], the SMT solver we use in ImPROV. In order to do so, we need to guide Z3 to return the exact number of counterexamples that we want. Consider that, for example, we want two counterexamples per rule validation. After Z3 returns the first counterexample, the algorithm adds the negation of this counterexample to the SMT formula and makes another SMT solver call, returning the second counterexample, as intended. We test ImPROV with five different numbers of counterexamples returned per validation call: one, two, five, ten and all counterexamples.

Table 5.3 presents the results of using each one of these possibilities over all nine instances. As expected, when the amount of counterexamples requested per iterations increases, the number of overall iterations of the main loop decreases. In all the applications (ER, RC and AR) the number of iterations decreases almost linearly when testing one, two, five and ten counterexamples returned per validation call. However, when all the existing counterexamples are generated, the number of iterations is always the same for each application. In other words, apart from the instance where the algorithm reaches the time limit (RC3), the instances from the ER application take 3 iterations to finish, the instances from the RC application (RC1 and RC2) take 10 iterations to finish and the instances from the AR application take 4 iterations to finish. Note that the number of iterations does not have a direct relation with the total time of the algorithm.

In the entity resolution (ER) application, the results show that the best configuration is to demand all counterexamples per rule validation. When asking for all the counterexamples available, the algorithm finishes much faster than the other configurations. In this case, the variation from one to ten counterexamples does not change the results significantly. The greatest difference is 64 seconds with ER3 (total time around 15 minutes). In this application, the grounded clauses obtained from the validation of the first rules cover the satisfiability of the remaining ones. In other words, by getting all the counterexamples of the first rules and, consequently grounding all those clauses, the rest of the constraints are not falsified by the MaxSAT solution. This particular aspect is then reflected in the number of iterations (3) and the number of grounded clauses. We analyse both these components later in the chapter. On the other hand, when the algorithm demands between one and ten counterexamples per validation, the MaxSAT solution is falsified by the remaining constraints throughout the computation.

In the relational classification (RC) application, the results show that the best number of counterexamples per rule validation is five. In these three instances (RC1, RC2 and RC3), the best amount of returned counterexamples per validation call is not as straightforward as in the previous test. The results from table 5.3 do not show all the test options from the RC application. We also use three and four as test options. However, the results from using both three and four counterexamples are worse than those presented in table 5.3 (using five counterexamples). The results from asking for all the counterexamples upfront are not better. The results from using this option are not close to the other four presented experiments. In this first two applications, the options of one, two, five and ten counterexamples per rule validation have similar results, while the option to ask for all counterexamples is always separate

Table 5.3: Results of ImPROV using a different number of counterexamples returned by the validation call. Experiments that timed out (denoted '-') exceeded the time limit. The instances used are the ones presented in table 5.1. (m = minutes, s = seconds)

	databases	# counterexamples per iterations	# iterations	total time (m = min, s = sec)
Entity Resolution	ER1	1	465	32s
		2	240	28s
		5	103	29s
		10	57	30s
		all	3	11s
	ER2	1	1029	3m14s
		2	525	3m02s
		5	224	3m16s
		10	118	3m13s
		all	3	38s
	ER3	1	1820	14m13s
		2	926	13m17s
		5	382	13m19s
		10	201	14m21s
		all	3	2m01s
Relational Classification	RC1	1	203	1m05s
		2	102	42s
		5	48	43s
		10	27	41s
		all	10	1m09s
	RC2	1	223	3m48s
		2	116	2m26s
		5	51	2m44s
		10	29	4m01s
		all	10	7m04s
	RC3	1	494	10m29s
		2	273	8m37s
		5	147	7m49s
		10	98	14m28s
		all	-	-
Advisor Recommendation	AR1	1	1445	1m32s
		2	728	1m26s
		5	289	1m31s
		10	145	1m40s
		all	4	48s
	AR2	1	3290	12m15s
		2	1641	12m20s
		5	664	11m57s
		10	329	13m51s
		all	4	3m56s
	AR3	1	4274	25m09s
		2	2138	24m44s
		5	858	25m35s
		10	432	27m04s
		all	4	6m28s

from the rest. In ER, asking for all countereexamples is a lot faster than the others, while in RC is a lot slower. Note that, the algorithm does not finish when testing RC3 (30 minutes time out). In this application, these results show us two different aspects: first, since the algorithm is much slower to finish

when asking for all the counterexamples (RC3 even reaches the time out limit), there is an excessive amount of counterexamples for at least one rule; secondly, since the other tests (one, two, five and ten counterexamples) finish in a timely manner, we can deduce that there is at least one counterexample (probably more) that cover all the others, i.e. by adding a few counterexamples, these few may cover the validation of many others.

In the advisor recommendation (AR) application, the results show that the best option is to ask for all the counterexamples upfront, each time there is a rule validation. As opposed to the previous tests, this option is, by far, the best one for solving these three instances (AR1, AR2 and AR3). Results show that the algorithm needs plenty of counterexamples in order to solve the problem. Hence, it is better to ask for them upfront. Moreover, if the algorithm returns between one and ten counterexamples per validation call the method becomes slow. In this case, the algorithm takes too long to finish due to two main reasons: first, the number of iterations is much larger; second, the unnecessary use of the MaxSAT solver in each iteration may hinder the computation. Note that, the final number of MaxSAT variables at the end of the algorithm is very similar both when the SMT call returns ten counterexamples per rule validation and when the SMT call returns all the counterexamples per validation call. Therefore, in the AR instances, the amount of counterexamples returned per rule validation does not modify the amount of counterexamples the algorithm needs overall. Requesting them all in advance is the best solution for this problem.

Following these mixed results (there is no obvious setting that is the best for all applications), we implemented a dynamic method to manage the amount of counterexamples returned per validation call. With this method each rule has an independent amount of counterexamples that is returned per rule validation. The initial amount of counterexamples is two, for every rule. Then, if all the counterexamples requested are returned, the amount is doubled for the next iteration. Otherwise, the amount of counterexamples is divided by two for the next iteration. This decrease happens when a rule is proved before all the counterexamples were returned. Important to note that the minimum number of counterexamples requested is two. For example, if rule r starts with two counterexamples and both are returned in the first iteration, then in the following iteration the algorithm requests four counterexamples and then eight and so on. The idea is that if the algorithm needs the counterexamples, the SMT solver may as well give them upfront. The results, however, showed a worse performance in both ER and RC. In order to simplify this section, we do not present these results in the document.

After some analysis, we determined that asking for dozens or even hundreds of counterexamples per validation call was not a good approach. Consequently, we developed a second version of the dynamic method: we added a maximum amount of counterexamples returned per validation call. Eight was the best upper bound. Hence, the amount of counterexamples returned by each rule validation could vary between two, four or eight, depending on the previous iterations. The results obtained with this approach were slightly better than the ones from the previous version. However, this dynamic method still had worse results than a simple constant two counterexample method. For the same reason as the other dynamic approach, we do not present these results in the document.

This configuration (amount of counterexamples) is the hardest. It can vary immensely with each in-

Table 5.4: Times obtained with ImPROV using the best possible configuration. The instances used are the ones presented in table 5.1. (m = minutes, s = seconds)

	databases	total time (m = min, s = sec)	SMT time		MaxSAT time	
			(m = min, s = sec)	% of total time	(s = sec)	% of total time
Entity Resolution	ER1	28s	27s	96.4 %	0.1s	0.36 %
	ER2	3m02s	2m57s	97.2 %	0.5s	0.27 %
	ER3	13m17s	13m00s	97.9 %	2.3s	0.29 %
Relational Classification	RC1	42s	41s	97.6 %	0.1s	0.24 %
	RC2	2m26s	2m24s	98.6 %	0.1s	0.07 %
	RC3	8m37s	8m17s	96.1 %	1.4s	0.27 %
Advisor Recommendation	AR1	1m26s	1m21s	94.2 %	2s	2.33 %
	AR2	12m20s	11m40s	94.6 %	21s	2.84 %
	AR3	24m44s	23m28s	94.9 %	43s	2.90 %

stance or application. The aforementioned Inference via Proof and Refutation (IPR) algorithm presented in section 3.3 demands all the counterexamples per validation call. In our case, despite the results from requesting all counterexamples per rule validation in the instances from ER and AR, we consider the use of two returned counterexamples per validation call the one with the best results overall. The main cause for the use of two counterexamples instead of all is the RC3 instance, where the experiment that returns all counterexamples per validation reaches the time out limit. We do not want to use any configuration that reaches the time out limit. Hence, we request two counterexamples per validation call throughout the rest of the evaluation chapter.

5.2.3 Incremental vs non-incremental MaxSAT

This section describes the possible configuration of line 15 of the ImPROV algorithm (algorithm 4.2.1). As previously explained, the weighted partial MaxSAT (WPMS) solver used can be any off-the-shelf MaxSAT solver. LBX [33] (a minimal correction subset (MCS) extraction algorithm) or the simplest non-incremental version of Open-WBO could be used as the underlying solver. However, ImPROV uses an incremental core-guided algorithm [47] that was developed specifically to solve sequential Maximum Satisfiability problems. This algorithm yields an average speedup of 1.8x over the non-incremental approach [47]. On some instances, the benefit is 4.7x. The overall performance is improved by reusing computation across similar MaxSAT instances in the same sequence.

5.2.4 Results with ImPROV's best configuration

Table 5.4 shows ImPROV's results in detail. Note that the best configuration of ImPROV is used to gather these results. As explained in the previous sections, the best configuration for ImPROV includes the use of an eager propagation in the beginning of the algorithm, requesting two counterexamples per validation call and using an incremental core-guided MaxSAT algorithm throughout the main loop.

There are two critical calls in our main loop: first, the MaxSAT solver call with all the current grounded

clauses; secondly, the SMT solver call, used to validate the rules. For each instance, we analyse how much time do both these critical calls take. The spare time that is not present in table 5.4 consists of handling the input, managing both the MaxSAT and the SMT formulas, performing the termination check after each iteration and some other minor details.

The results from all the instances suffer from the same problem. On average, across all the instances, 96.39% of the computation is spent in the SMT solver calls. These calls are vital to the algorithm, considering that they guide the search, that is to say, they decide which clauses are grounded and added to the MaxSAT formula. On the other hand, the algorithm spends merely 1.06% of the total time, on average, executing the MaxSAT calls. Even though there are some instances that have more than ten thousand MaxSAT variables, 2.9% is the maximum percentage that the MaxSAT calls take during the computation (AR3). As previously mentioned, MaxSAT solvers are the most researched and developed solvers when it comes to this kind of problem (certain objectives we want to optimize and some rules we do not want to break).

Considering these results, the clear bottleneck of the computation is the rule validation method. The use of an SMT solver may not be the best solution to validate rules after each MaxSAT solver call. We analyse this topic thoroughly in the following section.

5.3 Analysis of the ImPROV algorithm

This section presents the results from using Tuffy [37], the Inference via Proof and Refutation (IPR) algorithm [26] and Inference using PROpagation and Validation (ImPROV). The most important aspect of the results is the correctness of the solution, that is, if the solution is sound and optimal. Sound, meaning the solution does not violate any hard constraints and optimal, meaning the lowest possible solution cost. The second most important aspect is the total time of the computation. Note that, when we test the IPR algorithm, we use MiFuMaX [19] as the underlying MaxSAT solver. MiFuMaX is a sound and optimal MaxSAT solver, therefore the IPR algorithm presents sound and optimal solutions. When we test Tuffy, we use Maximum A Posteriori (MAP) inference. The instances used are the ones introduced in table 5.1. Table 5.5 shows, in detail, the outcome from the three algorithms.

5.3.1 Correctness of the algorithms

First of all, we evaluate the correctness of the algorithms. As expected, since both ImPROV and IPR are sound and optimal, they present the same solution for all nine instances. Both algorithms have the same positive literals in the nine solutions. These matched solutions are expected because of the similar algorithmic structure and how both groundings evolve during the computation. Important to note that both solutions can be different and correct. In other words, two distinct solutions can falsify different soft constraints and still have the same solution cost (sum of the weight of the falsified soft clauses). However, the fact that they are equal confirms the soundness and optimality of the ImPROV algorithm. On the other hand, the results show that Tuffy is not optimal. In the instances from the AR application,

Table 5.5: Results of evaluating TUFFY, IPR and ImPROV on three benchmark applications. Experiments that timed out (denoted '-') exceeded the time limit. The instances used are the ones presented in table 5.1.

	data bases	# iterations		# ground clauses			Solution cost			total time (m = min, s = sec)		
		IPR	ImPROV	TUFFY	IPR	ImPROV	TUFFY	IPR	ImPROV	TUFFY	IPR	ImPROV
Entity Resol.	ER1	3	240	27900	1119	1164	300	0	0	2m23s	3s	28s
	ER2	3	525	93150	2259	2320	809	0	0	13m05s	5s	3m02s
	ER3	3	926	55901	3849	3939	-	0	0	-	7s	13m17s
Relational Classif.	RC1	13	102	2161	1367	1653	18	0	0	5s	3s	42s
	RC2	13	116	3987	2488	3119	10	0	0	7s	4s	2m26s
	RC3	13	273	55901	6124	4758	633	0	0	55s	21s	8m37s
Advisor Recom.	AR1	3	728	1236	3240	9291	5314.8	3320	3320	5s	4s	1m26s
	AR2	3	1641	2956	7260	21216	12710.8	7260	7260	5s	6s	12m20s
	AR3	3	2138	3868	9316	27380	16632.4	9261.6	9261.6	6s	8s	24m44s

Tuffy is close to being correct. On average, Tuffy returns 97.7% of the positive literals present in the solution from ImPROV. However, in the instances from the other two applications (ER and RC), Tuffy has approximately half of the positive literals present in ImPROV's solution.

5.3.2 Number of iterations

Tuffy does not have the same algorithmic structure as IPR or ImPROV. Simply put, Tuffy does not work with iterations. Tuffy applies a non-iterative technique which is divided into a grounding phase and a solving phase. Thus, when evaluating the iterations of each algorithm, we just compare the results of IPR and ImPROV. Important to note that ImPROV uses two as the amount of counterexamples returned per validation, while IPR requests all existing counterexamples per validation. Considering the amount of counterexamples requested by each algorithm, the results are foreseeable. The ImPROV algorithm has a lot more iterations than the IPR algorithm, in every instance tested. This fact is not necessarily bad for any of the algorithms. A high number of iterations does not correlate directly to a good or a bad computation time, and neither does a low number of iterations. On average, the ImPROV algorithm takes 564 iterations to finish the ER instances, whereas the IPR takes 3 main loop iterations. The instances from the RC application take 13 iterations for IPR and 164 iterations for ImPROV, on average. The results from the last application (AR) show similar results: ImPROV has a high number of iterations (1502, on average) and IPR has a low number of iterations (3, on average).

5.3.3 Number of grounded clauses

At the end of the algorithm, the amount of grounded clauses is a statistic that can show if some of the computation may have been wasted. Our goal is to minimize the number of grounded clauses while maintaining soundness and optimality. Grounding fewer clauses suggests a more efficient algorithm, since it does not waste valuable time with inconsequential computation. Regarding the amount of grounded clauses by Tuffy, table 5.5 shows two distinct groups: first, the instances from the ER and

RC applications; secondly, the instances from AR. Since Tuffy generates an initial set of grounded constraints that is expected to be a superset of all the required grounded constraints, the results from the first group are as expected: Tuffy grounds more clauses than both the other algorithms in the six instances. In the second group, however, the quality of the solution is downgraded since the set of selected clauses to be eagerly grounded is too small. In the first five instances, IPR and ImPROV have a similar amount of grounded clauses. However, contrarily to what is expected, IPR grounds fewer clauses than ImPROV in all of the first five instances. It is counterintuitive that the algorithm demanding all the counterexamples per validation call grounds fewer clauses than the algorithm demanding only two counterexamples per rule validation, but, as introduced in section 5.2.2, the counterexamples found by the first rules cover the satisfiability of the rest. This is not the case in RC3. In RC3, the results match the presumed idea: ImPROV grounds fewer clauses than IPR. In this case, the grounding of the first rules do not cover the satisfiability of the remaining constraints, causing IPR to ground rules unnecessarily. In the instances from the AR application, the amount of grounded clauses are dissimilar. The three algorithms have distinct amounts of grounded clauses. As stated previously, in AR, Tuffy selects a set too small in order to do its eager ground. The main cause for the excessive amount of grounded clauses from ImPROV is the way the algorithm encodes the formulas from this specific MLN. These formulas are conjunctions of literals. This type of formula is not efficiently encoded in ImPROV since it uses too many auxiliary variables and clauses as explained in section 3.1.

5.3.4 Solution cost

The solution cost allows to identify the algorithms or solvers that are not optimal, i.e. do not return a solution that minimizes the sum of the weights of the falsified soft clauses (solution cost). In the instances from ER and RC, IPR and ImPROV have a solution cost of 0, meaning that the solution does not falsify any soft constraints. On the other hand, Tuffy presents solution costs higher than 0. These solution costs are expected since Tuffy is not optimal. In AR, ImPROV and IPR have the same solution cost across all three instances. These solution costs are higher than 0 since the optimal solution falsifies some soft constraints. Once again, Tuffy presents higher solution costs for all the instances from the AR application.

5.3.5 Performance of the algorithms

As explained in the beginning of this section, after the correctness of the solution is guaranteed, we evaluate the time each algorithm takes to finish the computation. Recall that unlike IPR and ImPROV, Tuffy is neither sound nor optimal. The results from the ER application show that Tuffy is the slowest method in all three instances. Moreover, in ER3, Tuffy reaches the time out limit. In these same instances, IPR is the fastest method, taking on average 5 seconds to solve each of the instances. The results from both the other applications (RC and AR) are similar to one another. Tuffy and IPR terminate the overall computation in under one minute in all of these instances (RC1, RC2, RC3, AR1, AR2 and AR3). They are both faster than ImPROV in these six instances.

Regarding the overall performance of ImPROV, there are two distinct reasons that slow down the computation. First, as explained in section 5.2.4, the validation of the rules, that is, the SMT solver calls are the bottleneck of the algorithm. This technology is not very good and simply underdeveloped when trying to get numerous counterexamples. Z3 is not currently capable of returning more than one counterexample per call. This limitation alone shows how rudimentary it is when applied with this purpose. Secondly, the amount of counterexamples requested per validation is not an easy value to configure. It depends on the examples and on the technology used. For IPR, for instance, asking for all the counterexamples upfront is the best configuration. With ImPROV, however, depending on the instance, requesting all the existing counterexamples in each validation call may slow the algorithm down significantly.

5.4 Summary

In conclusion, this chapter thoroughly explains the instances used in all the experiments, the possible configurations of the Inference using PROpagation and Validation (ImPROV) algorithm and an analysis of its results.

The applications (entity resolution, relational classification and advisor recommendation) that provide the instances are diverse and have both hard and soft constraints.

ImPROV's configuration is not straightforward and has many possible settings. The main decisions we explore are the use of an initial eager propagation, the amount of counterexamples returned per validation call and the use of an incremental MaxSAT solver instead of a more common, non-incremental MaxSAT solver.

Finally, we compare ImPROV with two state-of-the-art engines: Tuffy and IPR. Results show that IPR is the best engine. IPR returns correct solutions and it is the fastest to conclude the computation. With ImPROV, the use of an SMT solver to validate each rule is the main bottleneck of the algorithm. The validation calls slow the algorithm down considerably. On average, 96.39% of the computation is spent in the SMT solver calls. Nevertheless, ImPROV is better than Tuffy. ImPROV always returns a sound and optimal solution, while Tuffy does not guarantee neither soundness nor optimality of the solution.

Chapter 6

Conclusions

Markov Logic Network (MLN) combines two aspects that help depict logic problems: first-order logic and weights. In other words, MLN is a set of first-order logic formulas where each formula has an attached weight. Together with a set of facts that represent the domain of each relation, it can present a real world application or situation. First-order logic enables us to compactly represent a wide variety of knowledge [41]. It is also possible to use existential quantifiers to specify different types of circumstances. The attached weight from each formula reflects its importance. Furthermore, we can use the weight to indicate that a certain formula needs to be satisfied (hard constraint), that is, assigning an infinite weight to a formula.

Our main goal is creating an algorithm that returns correct answers. A correct answer consists of two traits. First, the solution must be sound: it can not violate any hard constraints. Secondly, the solution must be optimal: the algorithm must return a solution with the lowest possible cost. The following goal (less important) is having a better performance than the other state-of-the-art engines. In this case, performance is measured by the time each algorithm takes to conclude the problem. The sooner the algorithm ends the better. In this work, the tool Inference using PROpagation and Validation (ImPROV) was developed with these two main goals. Ultimately, we compare ImPROV with Inference via Proof and Refutation (IPR) [26] and Tuffy [37] over three real world applications: Entity Resolution (ER), Relational Classification (RC) and Advisor Recommendation (AR).

All in all, ImPROV is a combination of four important ideas. First, we use the structure of Horn constraints to perform an eager propagation using the facts provided from the input. This propagation reduces the number of iterations of the main loop, therefore accelerating the convergence of the algorithm. Secondly, unlike some of the recently presented engines to solve MLN, we use an Satisfiability Modulo Theories (SMT) solver (Z3 [10]) to validate the rules throughout the computation. The main difference between ImPROV and the other state-of-the-art algorithms is the use of an incremental core-guided Maximum Satisfiability (MaxSAT) algorithm that reuses information from the previous iterations, thus making the algorithm spend less time with the MaxSAT solver calls. This MaxSAT algorithm is the *Fu & Malik* algorithm with incremental SAT presented by Martins et al. [30]. Lastly, ImPROV terminates with soundness and optimality.

ImPROV's algorithmic structure is split into two distinct sections: the initializations and the main loop. The initializations contain the processing of the input, the conversion of all the rules and the initial eager propagation. Considering that we use a MaxSAT solver call at the beginning of each iteration, all the rules must be converted from first-order logic to Conjunctive Normal Form (CNF). Then, if all the conditions are met, ImPROV performs an initial eager propagation. The main loop always starts by calling a MaxSAT solver. In our case, we use the *Fu & Malik* implementation in the Open-WBO [31] MaxSAT solver. Next, we validate each rule using Z3 and ground the counterexamples found. At the end of each iteration, if all the rules that are not grounded are satisfied by the current MaxSAT solution, the algorithm returns the current MaxSAT solution. This solution is sound and optimal.

As previously stated, our first and most important goal with ImPROV was to get sound and optimal solutions to MLN problems. Unlike Tuffy, ImPROV accomplishes this goal in every MLN problem. Regarding ImPROV's performance, we notice that the configuration of the algorithm can alter the results immensely. Occasionally, the algorithm can be five times faster depending on the configuration. Nevertheless, it is clear that the bottleneck of the algorithm is the use of an SMT solver to validate the rules. After analysing ImPROV over all instances, on average, 96.39% of the computation is spent in the SMT solver calls. In comparison with the state-of-the-art engines, IPR has a better performance than ImPROV and Tuffy in every instance. ImPROV and Tuffy have some problems in specific applications so they present some mixed results between them. ImPROV is faster in the instances from the ER application, while Tuffy is faster in the remaining applications. In conclusion, IPR is the best engine out of the three since it provides sound and optimal solution and is also the fastest. ImPROV is better than Tuffy due to the quality of the solution. ImPROV always presents a sound and optimal solution, while Tuffy does not guarantee those two characteristics.

As future work, we propose to integrate the use of existential quantifiers in the MLN problems. $\exists x \quad Dog(x) \implies Owns(Pedro, x)$ is an example of a first-order logic formula that uses an existential quantifier. This formula represents that Pedro owns a dog. This kind of formulas can be useful when portraying logic problems. It permits users to say that Pedro owns at least one dog. Regarding the solution proposed, the future work is focused on the SMT solver calls, mainly because this is the section that bottlenecks the entire computation. In order to improve our rule validation method, we have two distinct options: first, we can replace the SMT solver with a different technology altogether. This technology could just validate a set of specifically selected rules (maybe only Horn constraints) or it could completely replace the SMT solver and validate all rules; secondly, we can improve how the SMT solver is used. In particular, the approach we use to encode the SMT formula. There may be a different and more efficient way to encode our method to obtain counterexamples. Moreover, improving the incrementality in the SMT solver can show great results since the SMT formulas are similar in structure from one iteration to the other. The better use of assumptions in the SMT solver can improve its solving technique, as well as the possible use of incrementality throughout the computation. The use of different theories and tactics can also enhance the performance of the SMT solver. Finally, since the validation of each rule is independent, we can parallelize the computation, i.e. split the validation of the rules into different cores.

Bibliography

[1] C. Ansótegui, M. L. Bonet, and J. Levy. Solving (weighted) partial maxsat through satisfiability testing. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 427–440. Springer, 2009. ISBN 978-3-642-02776-5. doi: 10.1007/978-3-642-02777-2_39. URL http://dx.doi.org/10.1007/978-3-642-02777-2_39.

[2] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009. ISBN 978-1-58603-929-5. doi: 10.3233/978-1-58603-929-5-825. URL https://doi.org/10.3233/978-1-58603-929-5-825.

[3] D. L. Berre and A. Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010. URL http://jsat.ewi.tudelft.nl/content/volume7/JSAT7_4_LeBerre.pdf.

[4] A. T. Chaganty, A. Lal, A. V. Nori, and S. K. Rajamani. Combining relational learning with SMT solvers using CEGAR. In N. Sharygina and H. Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 447–462. Springer, 2013. ISBN 978-3-642-39798-1. doi: 10.1007/978-3-642-39799-8_30. URL http://dx.doi.org/10.1007/978-3-642-39799-8_30.

[5] P. Chahuara, A. Fleury, F. Portet, and M. Vacher. Using markov logic network for on-line activity recognition from non-visual home automation sensors. In F. Paternò, B. E. R. de Ruyter, P. Markopoulos, C. Santoro, E. van Loenen, and K. Luyten, editors, *Ambient Intelligence - Third International Joint Conference, AmI 2012, Pisa, Italy, November 13-15, 2012. Proceedings*, volume 7683 of *Lecture Notes in Computer Science*, pages 177–192. Springer, 2012. ISBN 978-3-642-34897-6. doi: 10.1007/978-3-642-34898-3_12. URL http://dx.doi.org/10.1007/978-3-642-34898-3_12.

[6] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of

Lecture Notes in Computer Science, pages 154–169. Springer, 2000. ISBN 3-540-67770-4. doi: 10.1007/10722167_15. URL http://dx.doi.org/10.1007/10722167_15.

[7] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995. doi: 10.1007/BF00994018. URL http://dx.doi.org/10.1007/BF00994018.

[8] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 2(4):393–410, 1954. doi: 10.1287/opre.2.4.393. URL http://dx.doi.org/10.1287/opre.2.4.393.

[9] J. Davies, J. Cho, and F. Bacchus. Using learnt clauses in maxsat. In D. Cohen, editor, *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, volume 6308 of *Lecture Notes in Computer Science*, pages 176–190. Springer, 2010. ISBN 978-3-642-15395-2. doi: 10.1007/978-3-642-15396-9_17. URL http://dx.doi.org/10.1007/978-3-642-15396-9_17.

[10] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. ISBN 978-3-540-78799-0. doi: 10.1007/978-3-540-78800-3_24. URL http://dx.doi.org/10.1007/978-3-540-78800-3_24.

[11] T. Dierkes, M. Bichler, and R. Krishnan. Estimating the effect of word of mouth on churn and cross-buying in the mobile phone market with markov logic networks. *Decision Support Systems*, 51(3): 361–371, 2011. doi: 10.1016/j.dss.2011.01.002. URL http://dx.doi.org/10.1016/j.dss.2011.01.002.

[12] N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. ISBN 3-540-20851-8. doi: 10.1007/978-3-540-24605-3_37. URL http://dx.doi.org/10.1007/978-3-540-24605-3_37.

[13] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.

[14] Z. Fu and S. Malik. On solving the partial MAX-SAT problem. In A. Biere and C. P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, 2006. ISBN 3-540-37206-7. doi: 10.1007/11814948_25. URL http://dx.doi.org/10.1007/11814948_25.

[15] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *J. Autom. Reasoning*, 36(4):345–377, 2006. doi: 10.1007/s10817-006-9033-2. URL `http://dx.doi.org/10.1007/s10817-006-9033-2`.

[16] F. Heras, J. Larrosa, and A. Oliveras. Minimaxsat: An efficient weighted max-sat solver. *J. Artif. Intell. Res. (JAIR)*, 31:1–32, 2008. doi: 10.1613/jair.2347. URL `http://dx.doi.org/10.1613/jair.2347`.

[17] H. H. Hoos. An adaptive noise mechanism for walksat. In R. Dechter and R. S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada.*, pages 655–660. AAAI Press / The MIT Press, 2002. URL `http://www.aaai.org/Library/AAAI/2002/aaai02-098.php`.

[18] A. Horn. On sentences which are true of direct unions of algebras. *J. Symb. Log.*, 16(1):14–21, 1951. doi: 10.2307/2268661. URL `https://doi.org/10.2307/2268661`.

[19] M. Janota. Mifumax—a literate maxsat solver, 2013.

[20] H. A. Kautz, B. Selman, and Y. Jiang. A general stochastic approach to solving problems with hard and soft constraints. In D. Du, J. Gu, and P. M. Pardalos, editors, *Satisfiability Problem: Theory and Applications, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, March 11-13, 1996*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 573–586. DIMACS/AMS, 1996. URL `http://dimacs.rutgers.edu/Volumes/Vol35.html`.

[21] R. Kindermann and L. Snell. *Markov random fields and their applications*. 1980.

[22] S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, J. Wang, and P. Domingos. The alchemy system for statistical relational {AI}. 2009.

[23] M. Koshimura, T. Zhang, H. Fujita, and R. Hasegawa. Qmaxsat: A partial max-sat solver. *JSAT*, 8 (1/2):95–100, 2012. URL `http://jsat.ewi.tudelft.nl/content/volume8/JSAT8_6_Koshimura.pdf`.

[24] J. Larrosa, F. Heras, and S. de Givry. A logical approach to efficient max-sat solving. *Artif. Intell.*, 172(2-3):204–233, 2008. doi: 10.1016/j.artint.2007.05.006. URL `http://dx.doi.org/10.1016/j.artint.2007.05.006`.

[25] R. Mangal, X. Zhang, A. V. Nori, and M. Naik. A user-guided approach to program analysis. In E. D. Nitto, M. Harman, and P. Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 462–473. ACM, 2015. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786851. URL `http://doi.acm.org/10.1145/2786805.2786851`.

[26] R. Mangal, X. Zhang, A. Kamath, A. V. Nori, and M. Naik. Scaling relational inference using proofs and refutations. In D. Schuurmans and M. P. Wellman, editors, *Proceedings of the Thirtieth AAAI*

Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA., pages 3278–3286. AAAI Press, 2016. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12466.

[27] V. M. Manquinho, J. P. M. Silva, and J. Planes. Algorithms for weighted boolean optimization. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 495–508. Springer, 2009. ISBN 978-3-642-02776-5. doi: 10.1007/978-3-642-02777-2_45. URL http://dx.doi.org/10.1007/978-3-642-02777-2_45.

[28] J. Marques-Silva and V. M. Manquinho. Towards more effective unsatisfiability-based maximum satisfiability algorithms. In H. K. Büning and X. Zhao, editors, *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, volume 4996 of *Lecture Notes in Computer Science*, pages 225–230. Springer, 2008. ISBN 978-3-540-79718-0. doi: 10.1007/978-3-540-79719-7_21. URL http://dx.doi.org/10.1007/978-3-540-79719-7_21.

[29] J. Marques-Silva and J. Planes. On using unsatisfiability for solving maximum satisfiability. *CoRR*, abs/0712.1097, 2007. URL http://arxiv.org/abs/0712.1097.

[30] R. Martins, S. Joshi, V. M. Manquinho, and I. Lynce. Incremental cardinality constraints for maxsat. In B. O'Sullivan, editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 531–548. Springer, 2014. ISBN 978-3-319-10427-0. doi: 10.1007/978-3-319-10428-7_39. URL https://doi.org/10.1007/978-3-319-10428-7_39.

[31] R. Martins, V. M. Manquinho, and I. Lynce. Open-wbo: A modular maxsat solver,. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 438–445. Springer, 2014. ISBN 978-3-319-09283-6. doi: 10.1007/978-3-319-09284-3_33. URL https://doi.org/10.1007/978-3-319-09284-3_33.

[32] K. L. McMillan. Interpolation and sat-based model checking. In W. A. H. Jr. and F. Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003. ISBN 3-540-40524-0. doi: 10.1007/978-3-540-45069-6_1. URL http://dx.doi.org/10.1007/978-3-540-45069-6_1.

[33] C. Mencía, A. Previti, and J. Marques-Silva. Literal-based MCS extraction. In Q. Yang and M. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1973–1979. AAAI Press, 2015. ISBN 978-1-57735-738-4. URL http://ijcai.org/Abstract/15/280.

[34] A. Morgado, F. Heras, M. H. Liffiton, J. Planes, and J. Marques-Silva. Iterative and core-guided maxsat solving: A survey and assessment. *Constraints*, 18(4):478–534, 2013. doi: 10.1007/s10601-013-9146-2. URL http://dx.doi.org/10.1007/s10601-013-9146-2.

[35] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001. ISBN 1-58113-297-2. doi: 10.1145/378239.379017. URL http://doi.acm.org/10.1145/378239.379017.

[36] H. B. Newcombe, J. M. Kennedy, S. Axford, and A. P. James. Automatic linkage of vital records. *Science*, 130(3381):954–959, 1959.

[37] F. Niu, C. Ré, A. Doan, and J. W. Shavlik. Tuffy: Scaling up statistical inference in markov logic networks using an RDBMS. *PVLDB*, 4(6):373–384, 2011. URL http://www.vldb.org/pvldb/vol4/p373-niu.pdf.

[38] E. Oikarinen and M. Järvisalo. Max-asp: Maximum satisfiability of answer set programs. In E. Erdem, F. Lin, and T. Schaub, editors, *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*, pages 236–249. Springer, 2009. ISBN 978-3-642-04237-9. doi: 10.1007/978-3-642-04238-6_21. URL http://dx.doi.org/10.1007/978-3-642-04238-6_21.

[39] H. Poon and P. M. Domingos. Joint inference in information extraction. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 913–918. AAAI Press, 2007. ISBN 978-1-57735-323-2. URL http://www.aaai.org/Library/AAAI/2007/aaai07-145.php.

[40] H. Poon, P. M. Domingos, and M. Sumner. A general method for reducing the complexity of relational inference and its application to MCMC. In D. Fox and C. P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1075–1080. AAAI Press, 2008. ISBN 978-1-57735-368-3. URL http://www.aaai.org/Library/AAAI/2008/aaai08-170.php.

[41] M. Richardson and P. M. Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006. doi: 10.1007/s10994-006-5833-1. URL http://dx.doi.org/10.1007/s10994-006-5833-1.

[42] S. Riedel. Improving the accuracy and efficiency of MAP inference for markov logic. In D. A. McAllester and P. Myllymäki, editors, *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*, pages 468–475. AUAI Press, 2008. ISBN 0-9749039-4-9. URL https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=1966&proceeding_id=24.

[43] S. Riedel. Cutting plane map inference for markov logic. 2009.

[44] V. Ryvchin and O. Strichman. Local restarts. In H. K. Büning and X. Zhao, editors, *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, volume 4996 of *Lecture Notes in Computer Science*, pages 271–276. Springer, 2008. ISBN 978-3-540-79718-0. doi: 10.1007/978-3-540-79719-7_25. URL https://doi.org/10.1007/978-3-540-79719-7_25.

[45] B. Selman, H. J. Levesque, and D. G. Mitchell. A new method for solving hard satisfiability problems. In W. R. Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence. San Jose, CA, July 12-16, 1992.*, pages 440–446. AAAI Press / The MIT Press, 1992. ISBN 0-262-51063-4. URL http://www.aaai.org/Library/AAAI/1992/aaai92-068.php.

[46] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In B. Hayes-Roth and R. E. Korf, editors, *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1.*, pages 337–343. AAAI Press / The MIT Press, 1994. ISBN 0-262-61102-3. URL http://www.aaai.org/Library/AAAI/1994/aaai94-051.php.

[47] X. Si, X. Zhang, V. M. Manquinho, M. Janota, A. Ignatiev, and M. Naik. On incremental core-guided maxsat solving. In M. Rueher, editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 473–482. Springer, 2016. ISBN 978-3-319-44952-4. doi: 10.1007/978-3-319-44953-1_30. URL http://dx.doi.org/10.1007/978-3-319-44953-1_30.

[48] J. P. M. Silva and K. A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996. doi: 10.1109/ICCAD.1996.569607. URL http://dx.doi.org/10.1109/ICCAD.1996.569607.

[49] P. Singla and P. M. Domingos. Entity resolution with markov logic. In *Proceedings of the 6th IEEE International Conference on Data Mining (ICDM 2006), 18-22 December 2006, Hong Kong, China*, pages 572–582. IEEE Computer Society, 2006. ISBN 0-7695-2701-9. doi: 10.1109/ICDM.2006.65. URL http://dx.doi.org/10.1109/ICDM.2006.65.

[50] L. Snidaro, I. Visentini, and K. Bryan. Fusing uncertain knowledge and evidence for maritime situational awareness via markov logic networks. *Information Fusion*, 21:159–172, 2015. doi: 10.1016/j.inffus.2013.03.004. URL http://dx.doi.org/10.1016/j.inffus.2013.03.004.

[51] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011. doi: 10.5281/zenodo.16303.

[52] W. L. Winston, M. Venkataramanan, and J. B. Goldberg. *Introduction to mathematical programming*, volume 1. Thomson/Brooks/Cole Duxbury; Pacific Grove, CA, 2003.

[53] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *2003 Design, Automation and Test in Europe*

Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany, pages 10880–10885. IEEE Computer Society, 2003. ISBN 0-7695-1870-2. doi: 10.1109/DATE.2003.10014. URL `http://doi.ieeecomputersociety.org/10.1109/DATE.2003.10014`.

[54] X. Zhang, R. Mangal, R. Grigore, M. Naik, and H. Yang. On abstraction refinement for program analyses in datalog. In M. F. P. O'Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 27. ACM, 2014. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594327. URL `http://doi.acm.org/10.1145/2594291.2594327`.