Universitat de Lleida

Algorithms and efficient encodings for argumentation frameworks and arithmetic problems

Francesc Guitart Bravo

Dipòsit Legal: L.1694-2014
http://hdl.handle.net/10803/284474

ADVERTIMENT. L'accés als continguts d'aquesta tesi doctoral i la seva utilització ha de respectar els drets de la persona autora. Pot ser utilitzada per a consulta o estudi personal, així com en activitats o materials d'investigació i docència en els termes establerts a l'art. 32 del Text Refós de la Llei de Propietat Intel·lectual (RDL 1/1996). Per altres utilitzacions es requereix l'autorització prèvia i expressa de la persona autora. En qualsevol cas, en la utilització dels seus continguts caldrà indicar de forma clara el nom i cognoms de la persona autora i el títol de la tesi doctoral. No s'autoritza la seva reproducció o altres formes d'explotació efectuades amb finalitats de lucre ni la seva comunicació pública des d'un lloc aliè al servei TDX. Tampoc s'autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant als continguts de la tesi com als seus resums i índexs.

ADVERTENCIA. El acceso a los contenidos de esta tesis doctoral y su utilización debe respetar los derechos de la persona autora. Puede ser utilizada para consulta o estudio personal, así como en actividades o materiales de investigación y docencia en los términos establecidos en el art. 32 del Texto Refundido de la Ley de Propiedad Intelectual (RDL 1/1996). Para otros usos se requiere la autorización previa y expresa de la persona autora. En cualquier caso, en la utilización de sus contenidos se deberá indicar de forma clara el nombre y apellidos de la persona autora y el título de la tesis doctoral. No se autoriza su reproducción u otras formas de explotación efectuadas con fines lucrativos ni su comunicación pública desde un sitio ajeno al servicio TDR. Tampoco se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al contenido de la tesis como a sus resúmenes e índices.

WARNING. Access to the contents of this doctoral thesis and its use must respect the rights of the author. It can be used for reference or private study, as well as research and learning activities or materials in the terms established by the 32nd article of the Spanish Consolidated Copyright Act (RDL 1/1996). Express and previous authorization of the author is required for any other uses. In any case, when using its content, full name of the author and title of the thesis must be clearly indicated. Reproduction or other forms of for profit use or public communication from outside TDX service is not allowed. Presentation of its content in a window or frame external to TDX (framing) is not authorized either. These rights affect both the content of the thesis and its abstracts and indexes.

Universitat de Lleida
Departament d'Informàtica i Enginyeria Industrial

TESI DOCTORAL

Algorithms and Efficient Encodings for Argumentation Frameworks and Arithmetic Problems

Memòria de treball presentada per en **Francesc Guitart Bravo** a la *Universitat de Lleida* per a l'obtenció del títol de *Doctor en Informàtica.* El treball contingut en aquesta memòria ha estat realitzat sota la direcció de la Dra. **Teresa Alsinet Bernadó** i del Dr. **Ramón Béjar Torres**.

Lleida, maig de 2014

Als meus pares.

Contents

List of Figures

List of Tables

Abstract

In this thesis we focus on the design and implementation of a particular framework of *Possibilistic Defeasible Logic Programming* (RP-DeLP). This framework is based on a general notion of collective (non-binary) conflict among arguments allowing to ensure direct and indirect consistency properties with respect to the strict knowledge. An output of an RP-DeLP program is a pair of sets of warranted and blocked conclusions (literals), all of them recursively based on warranted conclusions but, while warranted conclusions do not generate any conflict, blocked conclusions do. An RP-DeLP program may have multiple outputs in case of circular definitions of conflicts among arguments.

We introduce two semantics, the first one where all possible outputs are computed and the second one which is a characterization of an unique output property. The computation of the outputs for both semantics relies on two main problems: the problem of finding a collective conflict among a set of arguments and the problem of finding *almost valid* arguments for a conclusion. Both problems are combinatorial problems, so we propose two resolution approaches: a first one based on *SAT* techniques and a second one based on *Answer Set Programming* techniques. We propose an implementation and we empirically test our algorithms. We provide an analysis on the performance of the implementation of the algorithms, and we explain the results on the resolution of some randomly generated problems.

In this thesis we also focus on the resolution of some combinatorial problems. We analyze, design and implement some resolution tools for arithmetic problems, modular constraints and networking problems. We studied empirically how our approaches perform and we compared them to other solving techniques known as best proposals in the literature.

Resum

Aquesta tesi doctoral se centra en el disseny i implementació d'un *framework* particular per *Possibilistic Defeasible Logic Programming* (RP-DeLP). Aquest framework es basa en una noció de conflicte col·lectiu (no binària) entre arguments que permet assegurar les propietats de consistència directa i indirecta respecte del coneixement estricte. Una sortida d'un programa RP-DeLP és una parella de conjunts de conclusions garantides i bloquejades (literals), totes elles basades recursivament en conclusions prèviament garantides. La diferència radica en què mentre les conclusions garantides no generen cap conflicte, les conclusions bloquejades sí que ho fan. Un programa RP-DeLP pot tenir múltiples sortides en el cas de definicions circulars de conflictes entre arguments.

S'introdueixen dues semàntiques pel sistema d'argumentació presentat. La primera d'elles pren en consideració totes les possibles sortides que poden ser obtingudes d'un programa RP-DeLP tenint en compte les diferents maneres de resoldre els conflictes circulars que poden sorgir. La segona semàntica se centra en el còmput d'una única sortida que està basada en la caracterització del que anomenem *maximal ideal output*. Aquesta sortida conté un nombre maximal de literals garantits, però que inclou només literals els arguments dels quals tenen els seus suports inclosos en la sortida.

El comput de les sortides per ambdues semàntiques es basa en la resolució de dos problemes principals: el problema de trobar conflictes col·lectius entre un conjunt d'arguments i el problema de trobar arguments *almost valid* per una conclusió. Ambdós problemes són considerats problemes combinatoris i es proposen dues aproximacions per a la resolució: una primera aproximació basada en tècniques *SAT* i una segona basada en *Answer Set Programming*. Es proposa una implementació i una anàlisi empírica dels algorismes implementats. Aquests algorismes es proven sobre un conjunt de problemes generats aleatòriament mitjançant un generador que permet la configuració dels diferents paràmetres dels problemes generats. Un cop obtinguts els resultats, s'estudia quina afectació han tingut els diferents paràmetres observant el temps de resolució i la informació obtinguda.

En aquesta tesi també s'estudien diferents tècniques de resolució per a altres problemes combinatoris. S'analitzen, dissenyen i implementen algunes eines de resolució per a problemes aritmètics, restriccions modulars i problemes de xarxes de comunicacions. S'ha estudiat com les aproximacions proposades es comporten en comparació amb altres tècniques proposades a la literatura considerades com les més eficients fins al moment.

Resumen

Esta tesis se centra en el diseño e implementación de un *framework* particular para *Possibilistic Defeasible Logic Programming* (RP-DeLP). Este framework está basado en la noción general de conflicto colectivo entre argumentos (no binario) que permite asegurar las propiedades de consistencia directa e indirecta respecto al conocimiento estricto. Una salida de un programa RP-DeLP es una tupla de conjuntos de conclusiones (literales) garantizadas y bloqueadas, todas ellas basadas recursivamente sobre conclusiones garantizadas con la particularidad de que mientras las conclusiones garantizadas no generan ningún conflicto, las conclusiones bloqueadas sí lo hacen. Un programa RP-DeLP puede tener múltiples salidas en el caso de que existan definiciones circulares de conflictos entre los argumentos.

Se introducen dos semánticas, la primera donde se computan todas las posibles salidas del programa y una segunda que nace de la caracterización de la propiedad de la salida única. El cómputo de las salidas para ambas semánticas se basa en la solución de dos problemas principales: el problema de la búsqueda de argumentos *almost valid* para una conclusión y la búsqueda de conflictos colectivos entre un conjunto de argumentos. Ambos problemas son problemas combinatorios y se proponen dos aproximaciones de resolución diferentes: una primera aproximación basada en técnicas *SAT* y otra segunda aproximación basada en técnicas de *Answer Set Programming.* Se propone una implementación y también se prueba empíricamente el comportamiento de los algoritmos propuestos. A través de un análisis sobre el comportamiento de la implementación se explican los resultados obtenidos. Para ello se generan problemas aleatorios donde algunas propiedades pueden ser controladas mediante la configuración de parámetros de entrada.

Adicionalmente esta tesis también se centra en la resolución de otros problemas combinatorios. Se analizan e implementan herramientas para la resolución de problemas aritméticos, restricciones modulares y problemas de redes de comunicaciones. Se propone un estudio empírico de las propuestas y se comparan con las aproximaciones, conocidas como más eficientes hasta el momento, de la literatura.

Acknowledgements

I would like to acknowledge all the people that helped me to develop the work contained in this document. I would like to express my sincerest thanks to:

- Ramón Bejar and Tere Alsinet, my advisors, for their support and for trusting me;
- the *Universitat de Lleida*, for giving me the opportunity to develop my research work through a PhD grant;
- the *Escola Politècnica Superior* and the *Departament d'Informatica i Enginyeria Industrial* at the *Universitat de Lleida* for providing me the necessary human and material resources;
- my colleagues Cèsar Fernández, Carles Mateu, Josep Maria Brunetti, Jordi Planes, Josep Argelich, Carlos Ansótegui and Alba Cabiscol for all the help provided;
- Felip Manyà, Lluís Godo and Antonio Morgado for the help provided and their wise advices;
- Chu Min Li and their colleagues, for all the help provided during my stay at the *Laboratorie du Modélisation, Information et Systèmes* at the *Université de Picardie Jules Verne*;
- to my family and friends for the wholehearted support, specially in the hardest moments;
- Bea, without her love and support, nothing of this would not have been possible.

1 Introduction

Contents

1.1. Context and Motivation

Argumentation plays a key role in Human reasoning. Everyday we share information among us by expressing ideas. Very often these ideas are related with many other ideas supporting or attacking a claim. We can say that those supports and attacks are part of an argumentative process within a dialogue. We can also see argumentation as an introspective process used when we face uncertainty or when we want to reason for a further understanding in a certain field of knowledge.

One of the main goals for Artificial Intelligence (AI) is to understand intelligent entities, but in contrast to other fields such as Philosophy or Psychology, which also are concerned to discover how humans think, AI tries to reproduce those intelligent behaviours in computers. The study of intelligence has been one of the oldest scientific research fields, but AI research was formally initiated in 1956, and from the beginning to nowadays, has turned one of the most biggest never ending puzzles in science.

Over the last ten years, argumentation has come to be a very important piece of study in the puzzle of AI. One of the most relevant contribution to this field is [Dun95a] where Dung stated a formalization for abstract argumentation frameworks. That framework is based on the definition of arguments and the relation between those arguments under the notion of attack. Arguments are evaluated using acceptability semantics where the attack relation is the core of all Dung's semantics.

There are a number of frameworks for modelling argumentation in logic. Logic-based formalization of argumentation frameworks has been extensively studied in recent years. One of the better-known approaches in this respect is Besanrd and Hunter's logic-based counterpart of Dung's theory [BH00], in which arguments are represented by a tuple where we distinguish the reasons, the conclusion and the method of inference by which the conclusion is meant to follow from the reasons. Common usual proposals for modelling argumentation in logic combine arguments for and against a particular conclusion. However, when we face knowledge in our everyday life we can not believe everything with the same level of certainty and sometimes the information we handle is not complete. To deal with potentially inconsistent an incomplete knowledge, there have been some formalisms to manage inconsistency and uncertainty in argumentation.

Defeasible Logic Programming (DeLP for short), formalized by García and Simari [GS04a], is one of such formalisms, combining results from defeasible argumentation theory and logic programming. Although DeLP has proven to be a suitable framework for building real-world applications that deal with incomplete and contradictory information in dynamic domains, it cannot deal with explicit uncertainty, nor with vague knowledge, as defeasible information is encoded in the object language using *defeasible rules.*

In [CSAG04], Chesñevar *et al.* define P-DeLP, a new logic programming language that extends original DeLP capabilities for qualitative reasoning by incorporating the treatment of possibilistic uncertainty and fuzzy knowledge.

In [ABG10], Alsinet, Béjar and Godo define a new recursive semantics for DeLP extended with weights for arguments and based on a general notion of collective (non-binary) conflict among arguments. In this framework, called *Recursive Possibilistic* DeLP (RP-DeLP for short), an output (or extension) of a program is a pair consisting of a set of warranted formulas and a set of blocked formulas with maximum strength. Arguments for both warranted and blocked formulas are recursively based on warranted formulas but, while warranted formulas do not generate any collective conflict, blocked conclusions do. Formulas that are neither warranted nor blocked correspond to rejected formulas. A key feature that this warrant recursive semantics addresses is the *closure under subarguments postulate* proposed by Caminada and Amgoud [Amg12], claiming that if an argument is excluded from an output, then all the arguments built on top of it should also be excluded from that output. Then, in case of circular definitions of conflict among arguments, the recursive semantics for warranted conclusions may result in multiple outputs for RP-DeLP programs.

The **first contribution** of this thesis is the design and implementation of a warrant procedure for computing the set of outputs that can be ultimately warranted in an RP-DeLP program. This warrant procedure is an algorithm which is based on the computation of two main queries for finding *valid arguments* and *finding collective conflicts*.

Due to the recursive characterization of warranted conclusions and that they can be naturally defined as sets computed with propositional rules and certain constrains, we designed and implemented two approaches for solving those queries:

- The first one based on the successful *SAT* encodings for solving STRIPS planning problems like the ones proposed by Kauz *et al.* in [KMS96, KS99a]. In a STRIPS planning problem, given an initial state, described with a set of predicates, the goal is to decide whether a desired goal state can be achieved by means of the application of a suitable sequence of actions.

- The second one is based on the *Answer Set Programming* (ASP for short) paradigm. We found the declarative programming paradigm ASP well suited for our purpose of finding arguments in the set of valid arguments and identifying arguments being part of a collective conflict due to the natural representation with propositional rules. In particular we reformulated SAT based constraints using the direct encoding proposed by Drescher and Walsh in [DW11a].

When considering the problem of deciding the set of conclusions that can be ultimately warranted in an RP-DeLP program, circular definitions of collective conflicts can arise. The usual skeptical approach would be to adopt the intersection of all possible outputs. However, in addition to the computational limitation, as stated by Pollock and Simari in [Pol09], adopting the intersection of all outputs may lead to an inconsistent output. Intuitively, for a conclusion, to be in the intersection does not guarantee the existence of an argument for it, that is recursively based on ultimately warranted conclusions.

The implementation of the corresponding designed algorithms for the computation of multiple an unique outputs for RP-DeLP programs, provides a powerful tool to derive information from potentially inconsistent knowledge bases. We used an imperative programming language such as Python to code the main algorithm and data structures. We also integrated modern SAT and ASP solvers to build an interpreter of RP-DeLP programs. That interpreter is able to read an RP-DeLP program and compute the set of warranted and blocked literals. One of the main advantages of having this interpreter running in a computer is that we were able to perform a series of experiments in order to empirically test how the RP-DeLP framework works. To do so, we developed a random instances generator. This generator can tune some of the main properties for generating random instances, among those properties are the length of the rules, number of variables, etc.

We also integrate the RP-DeLP interpreter into a web application. This application permits to any user in the world compute the outputs (choosing among the Maximal Ideal or multiple extensions) for an RP-DeLP program and also deciding if an ASP or an SAT solver is used to solve the queries in which the warrant procedure relies on. Another interesting feature of the web application is that we provide further information about the set of warranted and blocked information, including the supporting arguments for a blocked and a warranted literal.

The **second contribution** of this thesis is the study, design and implementation of efficient encodings for arithmetic and other combinatorial problems. Actual SAT solvers have become a very useful tool to consider when solving hard problems. Due to the high efficiency in the performance and the constant progress in solvers development, one might consider to represent the original problem as an SAT instance. The current high efficiency of SAT solvers turned SAT encodings a powerful tool for many practical industrial applications, such as Electronic Design Automation (EDA) and important problems in Artificial Intelligence, like STRIPS, that were originally believed to be problems not suitable for propositional logic satisfiability algorithms.

While most of the research in the SAT community has been focused on developing very efficient solvers to solve the SAT problem, there is also an interest in how problems are encoded efficiently. We studied three different problems and evaluated empirically how they perform in comparison with other solving tools:

1. Due to the high efficiency of modern SAT solvers, our interest was to encode well-known cryptographic problems as SAT formulas. Cryptographic systems rely on the hardness of reversing some arithmetic functions such as factorization and discrete logarithm. Previous articles have proposed integer factorization as SAT-benchmark, our goal is to introduce this benchmarks as the basis for new cryptographic SAT-benchmarks. We performed some empirical test and we compare with other problem solving techniques such as *pseudo-Boolean constraints* (PB constraints for short) and traditional algorithms for solving those problems. Our results indicate that these two problems are extremely hard for state-of-the-art SAT solvers, so they are good benchmarks for the research community interested in finding good SAT encodings for practical constraints.

2. There are more problems related with cryptographic functions. This is the case of linear modular arithmetic equations, apart from cryptography they are used in several interesting applications. As an example, in [GW97] linear modular arithmetic constraints are used to prune the search space in an algorithm for optimally solving bin

packing problems and in some frequency assignment problems, like in [CG93, NO06], frequency domains for sites are organized in groups that are congruence equivalence classes, so that adjacency constraints between pairs of frequencies can be defined with modular arithmetic. Our interest is to study modular arithmetic constraints on Boolean variables. We study and analyze some translations to PB constraints, which can be solved with native PB solvers. We also look at those PB solvers that have shown that a transformation to the SAT problem can be an effective solving strategy for PB problems.

3. There are several real world applications where SAT solving techniques are suitable for solving them. One of the best known is EDA but there are more topics where SAT is a very suitable tool. One of this topics is networking, to be more concrete, we focused on optical networking. All-optical networks are one of the most successful recent approaches to tackle with the need of high bandwidth, although they come with some inconveniences. One of such problems is the need to design networks that will be able to cope with existing and future demands with the least possible hardware deployment, especially without having to resort to costly frequency conversion or opto-electronic conversion.

   We focus on the *Routing Wavelength Assignment for Static Lightpath Establishment* (RWA-SLE for short), by encoding it as PB problem. Then we compare results using our solving method with other proposed approaches for a wide range of generated problem instances. Results show that, for those problems where it is hard to find a suitable set of routes and wavelength assignments our method performs better than other methods. Solving those hard instances is particularly interesting because, otherwise, more hardware deployment would be needed to meet the traffic requirements.

   Additionally to the study over the PB encodings for the RWA-SLE problem, we also present an Answer Set Programming solving approach. Answer Set programming has become an attractive tool for representation and reasoning. Although some solvers were proposed in [SNS02] [LPF$^+$06], recent work in ASP solving techniques such as conflict driven learning, backjumping, restart and watched literals lists has been proposed in [GKNS07], making Clasp the best performing ASP solver. This can bee seen in [GLN$^+$07] [DVB$^+$09] [CIR$^+$11] where Clasp and Potassco framework show very good performance in global results. Moreover Clasp also showed very good performances in SAT competitions, making ASP a good technique for CSP solving.

1.2. Contributions

The contributions of this thesis were obtained in two different parts. First, we devoted our efforts towards understand how to solve certain kind of combinatorial problems using SAT and ASP encodings. Secondly, we applied the knowledge obtained from the first phase to design and implement efficient solvers for the argumentation problems studied in the second part. In this section we summarize the contributions of this thesis. The contributions of the first part are:

- We presented SAT encodings for basic non-linear arithmetic operations: multiplication and modular exponentiation, used as the basic building blocks for encoding integer factorization and discrete logarithm as very challenging SAT instances. We performed an empirical study for comparing the performance of our encodings with other solving techniques.

- We studied how to solve efficiently pseudo-Boolean Modular constraints (a generalization of parity constraints). After this empirical study we have provided two new alternative approaches based in pseudo-Boolean constraints and SAT encodings. We studied the performance of our encodings.

- We studied the use of current SAT solvers for the resolution of all optical networks problems. Through a series of empirical studies, we proved that a good formulation allocates network resources more efficiently than other approaches based on greedy algorithms, extensively studied in the specialized literature, at least, for critically constrained problems. We also contributed with new pseudo-Boolean encoding variants, highly competitive with the existing SAT formulations for this problem, that make easier the task of extending the proposed formulation to other networking problems with different kind of constraints.

- We also performed an experimental investigation about the use of Answer Set Programming (ASP) solvers for the resolution of of all optical networks problems. We contributed with new Answer Set Programming encodings for the problem, and we studied how the tightness and the number of generated loop nogoods relate with the complexity of solving the instances. In addition, a comparison with the best previous pseudo-Boolean encoding has shown that the Answer Set Programming approach is quite competitive with the well established approach based on pseudo-Boolean solvers that use SAT encodings and state-of-the-art SAT solvers.

Then, once we learned about how to solve hard combinatorial problems with SAT and ASP encodings, we tackled the problem of designing and implementing efficient algorithms for different argumentation problems:

- We studied, designed and implemented algorithms that compute the outputs for a Recursive Possibilistic Defeasible Logic Programming (RP-DeLP) argumentation framework, but considering two different semantics:

  1. A semantics where the answer for a single argumentation problem instance can include multiple outputs, where an output contains a set of warranted and blocked conclusions and their arguments, given that mutually conflicting arguments are resolved by considering different outputs. Each output is guaranteed to satisfy certain consistency and closure properties.
  2. A semantics where there is an unique output for a single argumentation problem instance, called maximal ideal output, where the output represents the maximal possible consensus that is free from mutual conflicts between arguments but it also satisfies similar closure properties like in the previous semantics.

  After the design of a first version of algorithms for these two semantics, we subsequently designed refined versions that allow a more efficient implementation, because avoid the computation of certain structures and not necessary arguments with the goal of improving the overall performance.

- We contributed with the design and implementation of efficient encodings, based on SAT and ASP techniques, of the two main queries that are needed to solve by our argumentation algorithms: computing valid arguments and computing conflicting sets of arguments. These queries are needed by our algorithms under the both semantics considered.

- We designed and implemented a web system for the easy use of our argumentation algorithms, that allows the use of both the SAT encodings and the ASP encodings. The web system not only gives to the user the outputs generated from the two different semantics considered, but it also gives additional information to the user that allows to understand the arguments finally warranted and blocked in each output, with the goal of helping non-expert users on argumentation to better understand the outputs generated under a particular semantics.

- We empirically studied the performance of the algorithms when solving random problem instances. We generated sets of random problem instances that differ on some of the features, such as number of variables, number of clauses, and other structural characteristics. We also empirically studied and compared how the SAT and ASP encodings proposed allow the argumentation algorithms to scale up with problem size for some particular cases.

1.3. Publications

Some of the contributions done during this thesis have already been published in conferences and journals. We next list in chronological order those publications:

- Béjar, R., Fernández, C., and Guitart, F. Encoding Basic Arithmetic Operations for SAT-Solvers. In *Proceedings of CCIA2010. Pages 239 - 248.*

- Ansótegui, C., Béjar, R., Fernández, C., Guitart, F., and Mateu, C. Solving Pseudo-Boolean Modularity Constraints. In *Proceedings of ECAI2010. Pages 867 - 872.*

- Alsinet, T., Béjar, R., Godo, L., and Guitart, F. Maximal Ideal Recursive Semantics for Defeasible Argumentation. In *Proceedings of SUM2011. Pages 96-109.*

- Béjar, R., Fernández, C., Guitart, F., and Mateu, C. Towards an Efficient Use of Resources in All-Optical Networks. In *Proceedings of CCIA2011. Pages 61 - 70.*

- Alsinet, T., Béjar, R., Godo, L., and Guitart, F. Using Answer Set Programming for an Scalable Implementation of Defeasible Argumentation. In *Proceedings of ICTAI2012. Pages 1016 - 1021.*

- Alsinet, T., Béjar, R., Godo, L., and Guitart, F. On the Implementation of a Multiple Outputs Algorithm for Defeasible Argumentation. In *Proceedings of SUM2013. Pages 71 - 77.*

- Béjar, R., Fernández, C., Guitart, F., and Mateu, C. Solving Routing and Wavelength Assignment problem with Conflict-Driven ASP solvers. In *Proceedings of CCIA2013. Pages 60 - 63.*

- Alsinet, T., Béjar, R., Godo, L., and Guitart, F. Web Based System for Weighted Defeasible Argumentation In *Proceedings of CLIMA XIV. Pages 155 - 171.*

- Alsinet, T., Béjar, R., Godo, L., and Guitart, F. RP-DeLP: A Weighted Defeasible Argumentation Framework Based on a Recursive Semantics. In *Journal Of Logic And Computation*

- Alsinet, T., Béjar, R., Fernández, C., Guitart, F., and Mateu, C. Solving Routing and Wavelength Assignment problem with Conflict-Driven ASP solvers. In *AI Communications*.

2 SAT Encodings

In this chapter we summarize the work done to understand techniques for solving combinatorial problems. We focus this work on theoretical and practical problems. First we give a short introduction to SAT algorithms and then we move to three main problems: Arithmetic Problems, Modular Constraints and the Routing and Wavelength Problem for All-Optical Networks. For each problem we propose some encodings which are evaluated through a comparison to other problem solving techniques, and then we discuss the results obtained.

Contents

2.1. Introduction

The Boolean Satisfiability Problem (SAT) has received a lot of attention since the early 60'. SAT is a central problem in Computer Science, having both theoretical and practical interest. From a theoretical point of view, it was the first problem to be shown NP-complete and, from a practical point of view, it has very efficient algorithms (SAT solvers) which are publicly available for an easy use for the SAT community. In the past few years, the SAT community has worked hard to improve the performance of SAT solvers. As an example, a SAT competition is yearly organized, where researchers around the world present their improvements on this field.

Despite the fact that, unless $P = NP$, the time complexity SAT problem is know to be exponential in worst case, it is used in many areas such as software and hardware verification, planning , scheduling , cryptography. SAT solvers are becoming a suitable tool to tackle more and more practical problems, and instead of developing a search procedure for an specific problem domain it is worth to reformulate the original problem as SAT instance.

SAT solvers can be divided in two categories, complete and incomplete. On the one hand, complete methods can give a solution for a given input SAT formula or prove that the formula is unsatisfiable. On the other hand, incomplete methods can find a solution for a very large formula, but they fail to detect unsatisfiability when no solution exists. In the field of complete methods, most of the actual solvers are wise variations of the DPLL procedure which was already presented in the 60'. Next we will give a brief problem definition and some notation which will be used in the rest of the document.

A Boolean formula is a logical expression defined over variables that can take only two truth values: FALSE (0) and TRUE (1). A literal l is either a variable b or its negation $\neg b$. We are interested in a certain kind of formulas, which are in Conjunctive Normal Form (CNF). In a CNF formula, a clause c is a disjunction of literals, $l_1 \vee l_2 \cdots \vee l_n$ and the formula is a conjunction of clauses $c_1 \wedge c_2 \cdots \wedge c_n$.

A truth assignment to a set V of Boolean variables is a function σ assigning a truth value to a set of variables. We say that a formula F has a satisfying assignment if it is evaluated to 1 under a certain assignment. We say then that F is satisfiable.

A partial assignment for a formula F is a truth assignment to a subset of the variables of F. For a partial assignment ρ for a formula F, $F\mid_\rho$ denotes the simplified formula obtained by replacing the variables appearing in ρ with their specified values, removing all clauses with at least one TRUE literal, and deleting all occurrences of FALSE literals from the remaining clauses. If an assignment satisfies a formula F, this assignment is called a *model* of F.

The SAT problem of a formula F in CNF, consists in determining whether there exists an assignment to the Boolean variables of F that satisfies the formula.

The empty clause will be denoted as $\perp$, which is the clause with no literals and is unsatisfiable. A clause with only one literal is referred as unit clause. A clause with two literals is refered as binary clause.

2.1.1. Complete Methods

A complete method for a SAT formula F, either finds a solution for F or produces the empty clause ($\perp$) , meaning that F has no solution. The most widely used methods in complete solvers are variations over the DPLL algorithm. As we will see later, this algorithm performs backtracking on the search space of partial truth assignments and performs efficient pruning on falsified clauses. Modern SAT solvers add other techniques to this procedure, such as clause learning and watched literals lists.

DPLL Procedure

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm [DLL62] is a refinement of the previous Davis-Putnam algorithm, which was a resolution-based algorithm proposed by Davis an Putnam in 1960. The DPLL algorithm is a search process for finding a satisfying assignment for a given CNF formula, or proving its unsatifiability.

The backtracking algorithm runs by selecting an unassigned variable l and recursively search for a satisfying assignment for F. We say that the algorithm is in a branching step every time a literal is chosen. A decision is an assignment of the literal l to FALSE or TRUE in a branching step. For a partial assignment of F, we say that F is violated by σ if it contains the empty clause. The unit propagation procedure assigns the TRUE value to unit clauses in order to increase the efficiency of the global procedure.

Other Features of Modern DPLL-Based SAT Solvers

Apart from the DPLL algorithm, modern SAT solvers introduce many other techniques for handling SAT instances. It is not easy to keep an updated list of all those features, but we will give some explanation of the most used ones.

- *Decision strategy* is one of the techniques that most differs from a solver to another in variable selection. Some of the solvers use random assignment to the variables, while others make use of objective functions to maximize the current variable or clause state.

- *Clause learning* plays a critical role in solvers performance. It is used to prune the search in a different part of the space search by giving an explanation for failure points.

- *Non-chronological backtracking* forces the algorithm to analyze the conflict reason and then determines which settings of the current partial assignment are responsible for the conflict. Then it also determines how far has to backtrack to unassign variables being part of the conflict reasons.

- *Restarts* clear the decision stack after some backtracks. Instead of performing a long search, it splits the decision sequences in many shorter searches. It reduces the probability of the procedure to get stuck in an abnormally long search time and increases the availability of good splitting decisions by allowing the search to explore only the smaller subtrees under a decision node.

- *Watched literals* are data structures which maintain information of some special clauses in order to trigger fast unit propagation. The idea behind is that for each clause there are two watched literals, when a watched literal has an assignment the clause is reviewed, if there are more than one variables unassigned, then the pointer to the watched literal is updated to a new unassigned variable, but if there is only one unassigned variable left, is time to run unit propagation. This may led bigger usage of memory but speeds up the performance of the solver.

SAT Solvers

As explained later, we are concerned not only with solving problems in an efficient way, but with using the information provided by SAT solvers to get further information based on that results. That is the case for argumentation systems encodings, where an UNSAT instance verifies the absence of conflicts in a knowledge database. This is only one of the reasons why we need complete methods to solve most of our generated SAT instances. We used basically two main SAT solvers: Minisat and Precosat. Follows a description of its main features:

- *Minisat* is one of the most popular SAT solvers in the community. Its minimalistic and open-source nature makes it the starting point for many researchers and developers in SAT community. Apart of being a good learning tool, it was awarded in the 2005 SAT competition, being this a proof of its good efficiency. Minisat has been an excellent tool for us to solve SAT instances as well as to modify and implement some other features in our algorithms, such as unit propagation. Some key features of Minisat are:

    - *Easy to modify*, as it provides an excellent documentation and it is clearly programmed, being very easy to understand and also to modify.

    - *Highly efficient*, results show that it can solve SAT instances very efficiently.
    - *Designed for integration*, as minisat supports incremental SAT and has mechanisms for adding non-clausal constraints. It is a good choice for being integrated as a backend to another tool.

- *Precosat.* Precosat version 236 was the winner of the application track of the SAT competition in 2009. Newer versions have been submitted to more recent competitions.

2.1.2. Incomplete Methods

An incomplete method for solving the SAT problem is one that does not provide the guarantee that it will eventually either report a satisfying assignment or prove that the formula is unsatisfiable. Unlike the complete methods that do an exhaustive branching and backtracking search over the space of the problem, incomplete methods are generally based on stochastic local search.

The most known local search methods in SAT are GSAT [SLM92] and WalkSAT [SKC94]. By contrast, these procedures are able to solve hard instances with more variables than complete methods, though completeness is lost. Local search methods start typically with some ramdomly generated complete assignment and try to find a satisfying assignment by iteratively changing the assignment of one propositional variable. Each change of the assignment of a variable is called a variable flip, and variables are selected heuristically. Such changes are repeated until either a satisfying assignment is found or a pre-set maximun number of changes is reached.This process is repeated as needed, up to a pre-set number of times. This allows to explore the search space moving from one search space position to a neighboring position. The decision on each step (change) is based on information about local neighborhood only. Usually, local search algorithms do not explore the entire search space, and a given assignment may be considered more than once.

The main difference among the different local search algorithms for SAT lies in the strategy used to select the variable to be ipped next. Furthermore, local search algorithms can get trapped in local minimal and plateau regions of the search space, leading to premature stagnation of the search. One of the simplest mechanisms for avoiding premature stagnation of the search is random restart, which reinitializes the search if no solution has been found after a fixed number of steps. Random restarts are used in almost every local search algorithm for SAT.

2.2. Arithmetic Problems

Given the current success of using SAT encodings for solving problems with linear arithmetic constraints, see for example [ES06], our motivation is to start an investigation for finding the best we can do with SAT encodings for solving problems with non-linear arithmetic constraints. We consider two of such problems, integer factorization and discrete logarithm over a finite cyclic group, that are basic problems for cryptographic applications.

These two problems are also interesting from the point of view of artificial intelligence and constraint programming, as they are problems in the complexity class NP. Actually they are believed not to be polynomially solvable, they seem not to be NP-complete. So they are one of the few natural problems to be located somewhere between P and NP-complete problems, a class of problems not widely studied by these research communities.

We present SAT encodings for the basic building functions that can be used to define these problems: adders and multipliers. Our SAT encodings are based on Boolean circuit representations of such functions, following the best approaches found so far for encoding linear constraints with SAT.

Definition 2.1 (Boolean circuit)**.** *For every $n, m \in N$, a Boolean circuit C with n inputs and m outputs is a directed acyclic graph. It contains n nodes with no incoming edges; called the input nodes and m nodes with no outgoing edges, called output nodes. All other nodes are called gates and are labeled with one $\vee$, $\wedge$ or $\neg$ (the logical operations OR, AND, and NOT). The $\vee$ and $\wedge$ nodes have fanin (i.e., number of incoming edges) of 2 and the $\neg$ nodes have fanin 1. The size of C, denoted by $|C|$, is the number of nodes in it. The circuit is called a Boolean formula if each node has at most one outgoing edge.*

It is well known that each propositional formula can be converted to an equisatisfiable formula in CNF, e.g., by using Tseitin's encoding [Tse68]. The idea behind Tseitin transformations is to introduce a new variable x_G for each subformula $G = H_1 \circ H_2$ in F, where $\circ$ denotes any of the operators $\wedge$,$\vee$,$\rightarrow$ or $\leftrightarrow$, and we use $=$ to denote syntactic equality. The formula F is then translated into conjunctive normal form by adding a set of clauses $Tr(G)$ for each subformula G which enforces that the truth value of X_G is computed correctly given the truth values of X_{H_1} and X_{H_2}.

For the multiplication function, we also consider a translation to pseudo-Boolean linear equations, and a subsequent transformation to SAT, using current SAT encodings for such equations.

We compare the performance of our SAT-based approaches with the best algorithms for such problems: Quadratic Sieve factorization and Pollard's Rho discrete logarithm algorithms. Our results indicate that the performance of the SAT encodings is worse than for the current best specialized algorithms, thus indicating that these problems are interesting benchmarks for discovering efficient SAT encodings of practical constraints.

Nevertheless, we observe that the gap at performance between SAT encodings and specialized algorithms becomes narrower for discrete logarithm problems than for factorization. This opens a new line of research towards elliptic curve cryptography, where we expect to narrow the gap, given the state of the art of specialized elliptic curve algorithms.

Our main goal is to encode all the circuits as equisatisfiable formulas, so as long as there exists a model for the input variables also a model must exist for the outputs, and vicevercesa.

2.2.1. Factorization

Factorization is one of the two cornerstones where RSA security relies on. RSA [RSA78] is the first public-key cryptography schema suitable for digital signature as well as for encryption, being still widely used today. RSA security is based on the computational unfeasibility to factorize a public large integer, $n = x \cdot y$, where their prime factors, x and y, require certain conditions in order to avoid specific attacks that could discover x and y easily. Practical RSA public keys, nowadays consist of integers n with a length of a few thousands of bits.

To solve the factorization problem using a SAT solver, we will encode the multiplication problem as a CNF formula. So, given an encoding for the multiplication able to assign a value to variables x and y, the SAT solver will search for a proper assignment to the variable x, but if we encode it wisely we will be able to assign a value to variable x and the SAT solver will find an appropriate assignment for x and y such that it is the solution for the factorization problem.

Problem Definition

The decomposition of an integer $n = p_1^{e_1} \cdot p_2^{e_2} \dots p_k^{e_k}$, being p_i primes and e_i naturals, is unique. When we talk about n factorization we mean the problem of finding a non trivial decomposition for n. In this work we consider the special case of finding the non-trivial decompositions for an integer n that is the product of two primes x and y, where $x > 1$ and $y > 1$.

Adders

An adder is a circuit which takes two integers as input and outputs an integer which is the sum of the inputs. The simplest circuit performing this task is a half adder, which can represent the sum and the overflow (carry) of an input of two bits. A full adder can represent the sum and the overflow of an input of three bits representing two input bits plus the carry-in bit.

- **Half Adder:** A half adder needs two input bits (X and Y) and two output bits. The carry (C) bit is set to 1 only when the sum (S) can not be represented with a single bit.

$$\begin{array}{l} S \leftrightarrow X \oplus Y; \\ C \leftrightarrow X \wedge Y, \end{array}$$

- **Full Adder:** A Full Adder is a circuit capable of representing both the carry-in and the carry-out overflow in an arithmetic sum. It has three input variables A, B and C_{in} and two output variables S and C_{out} which represents the sum and the carry-out.

$$\begin{array}{l} S \leftrightarrow A \oplus B \oplus C_{in}; \\ C_{out} \leftrightarrow (A \wedge B) \vee (C_{in} \wedge (A \oplus B)), \end{array}$$

Multipliers

- **Array Multiplier:** For this implementation we used the propositional logic model proposed in [FMM03]. It can be implemented as an array of half and full adders. Using this schema, two l-bits length numbers can be multiplied using an array with l rows and $2l$-1 columns. For the encoding, as well as for Figure 2.1, we use X and Y as input variables and P as output. Figure 2.1 shows the array multiplier for $l = 4$.

$$\begin{array}{ll} I_{i,j} \leftrightarrow X_i \wedge Y_j & i,j = 0 \dots l-1 \\ S_{0,j} \leftrightarrow I_{0,j+1} \oplus I_{j+1,0} & j = 0 \dots l-2 \\ S_{i+1,j} \leftrightarrow C_{i,j} \oplus S_{i,j+1} \oplus I_{j+1,i+1} & i,j = 0 \dots l-2 \\ C_{0,j} \leftrightarrow I_{0,j+1} \wedge I_{j+1,0} & j = 0 \dots l-2 \\ C_{i+1,j} \leftrightarrow (I_{j+1,i+1} \wedge C_{i,j}) \vee & \\ (I_{j+1,i+1} \wedge S_{i,j+1}) \vee (C_{i,j} \wedge S_{i,j+1}) & i,j = 0 \dots l-2 \\ P_0 \leftrightarrow I_{0,0} & \\ P_i \leftrightarrow S_{i-1,0} & i = 1 \dots l-1 \\ P_{i+l} \leftrightarrow S_{l-1,i} & i = 0 \dots l-2 \\ P_{2l-1} \leftrightarrow C_{l-1,l-2} & \end{array}$$

- **Booth Multiplier:** Booth multiplication is a technique whereby x and y may be multiplied following a few simple steps [Boo51]. We decided to implement such algorithm because the corresponding circuit representation is smaller than for the array multiplier in terms of size.

  The Booth multiplier works with an iterative process, where the number of iterations is determined by the size of the multiplicand y, and where a sum of partial products is iteratively updated until it finally contains the value of the multiplication. First of all, let's define our

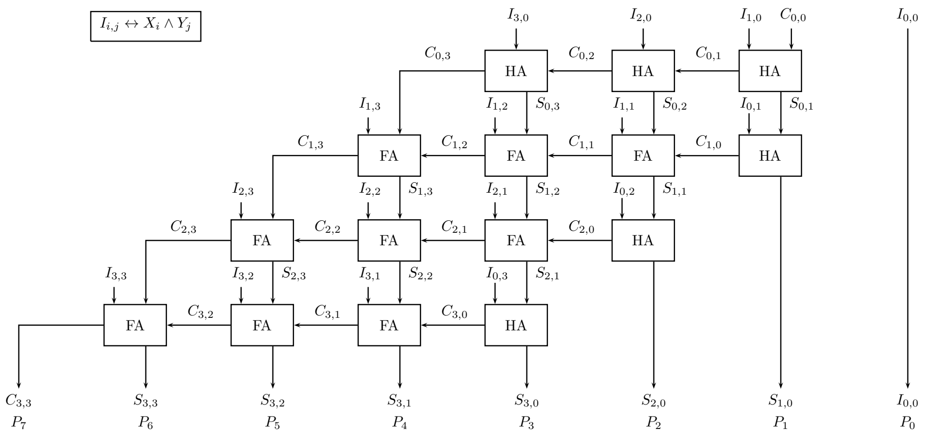


Figure 2.1: Modular design of an array multiplier.

variables. The input variables are the bit vectors multiplicands X, of size l_x, and Y, of size l_y, which contains the binary representation of x and y respectively. We define the size of the output as $l = l_x + l_y + 1$, and define the following additional bit vectors:

1. A is a bit vector of size l and contains the value of x on its l_x left-most bits, being the remaining $(l_y + 1)$ bits filled with zeros.
2. S is a bit vector of size l and contains the value of $-x$ (2-complement representation) on its l_x left-most bits, being the remaining $(l_y + 1)$ bits filled with zeros.
3. P is a bit vector of size l and it initially contains: on its l_x left-most bits the value 0, on the next l_y bits the value of y and in the final (right-most) bit the value 0. It represents a sum of partial products that at the end of the algorithm will contain the desired product of x and y.

The Booth multiplication process iterates the next steps l_y times, so at the end the bit vector P will contain the multiplication of x and y. At iteration i, the steps are:

1. If $P_0 = 0, P_1 = 0$, multiply the existing sum of partial products (P) by 2^{-1}. It is easy to see that this is an one place arithmetic shift to the right.
2. If $P_0 = 0, P_1 = 1$, add S to P and multiply by 2^{-1}.
3. If $P_0 = 1, P_1 = 0$, add A to P and multiply by 2^{-1}.
4. If $P_0 = 1, P_1 = 1$, multiply the sum of partial products by 2^{-1}.

Finally we drop the rightmost bit of P, so the product can be found on the $l_x + l_y$ left-most bits of P.

According to this procedure, we have implemented a SAT encoding based on its circuit representation, where the basic building functions are full and half adders, plus some additional small circuits that control the right action to perform at each iteration of the above iterative process. The SAT encoding uses the sets of Boolean variables $\{A_i, S_i | 0 \leq i \leq l-1\}$ and $\{P_{i,j} | 0 \leq i \leq l_y + 1, 0 \leq j \leq l-1\}$. The first set represents the bits of the vectors A and S, and the second one represents the value of the vector P at the different iterations of the algorithm; so $P_{i,j}$ represents the value of bit j of P at iteration i. Iteration 0 refers to the initial value of P. Next, we present the clauses of the SAT encoding. For enforcing the values of the bit vectors A and S, we add unitary clauses that set the value of variables A_i and S_i as described before. Similarly, the initial value of P will be set on the variables $P_{0,j}$. The rest of clauses of the encoding simulate the different iterations of the algorithm, so we have a similar set of clauses for each iteration i of the algorithm. We use additional sets of variables for simulating the computations performed at iteration i: $\{T_i | 1 \leq i \leq l_y\}$, $\{Q_{i,j} | 1 \leq i \leq l_y, 0 \leq j \leq l-1\}$, $\{C_{i,j} | 1 \leq i \leq l_y, 0 \leq j \leq l-1\}$. The set of clauses for iteration i are described below.

$$\begin{array}{ll}
T_i \leftrightarrow P_{i,0} \oplus P_{i,1} & \\
Q_{i,j} \leftrightarrow T_i \wedge ((P_{i,0} \wedge A_j) \vee (P_{i,1} \wedge S_j)) & j = 0 \ldots l-1 \\
C_{i,0} \leftrightarrow Q_{i,0} \wedge P_{i,0} & \\
P_{i+1,j-1} \leftrightarrow Q_{i,j} \oplus P_{i,j} \oplus C_{i,j-1} & j = 1 \ldots l-1 \\
C_{i,j} \leftrightarrow ((Q_{i,j} \wedge P_{i,j}) \vee (Q_{i,j} \wedge C_{i,j-1}) \vee (P_{i,j} \wedge C_{i,j-1})) & j = 1 \ldots l-1 \\
P_{i,l-1} \leftrightarrow P_{i,l-2} &
\end{array}$$

Pseudo Boolean Encoding

A linear *pseudo-Boolean constraint* (PB constraint) over Boolean variables is defined by $\sum_i c_i \cdot l_i \rhd p$ where c_i, the coefficients, and p, are integer constants, l_i are literals and $\rhd$ is one of the operators of $\{=, <, \leq, >, \geq\}$. Without loss of generality, these constraints can be rewritten to use the $\geq$ operator and positive coefficients (notice that $-c_i \cdot b_i$ can be rewritten as $c_i \cdot \neg b_i - c_i$). A coefficient c_i is said to be activated under a partial assignment if its corresponding literal l_i is assigned to true. Assuming that $\rhd$ is the $\geq$ operator, a pseudo-Boolean constraint is said to be satisfied under an assignment to its Boolean variables if the sum of its activated coefficients exceeds or is equal to p.

Consequently, the factorization problem can be encoded as follows:

$$\sum_{i=0}^{l-1} x_i 2^i \geq 2$$
$$\sum_{j=0}^{l-1} y_j 2^j \geq 2$$
$$\sum_{i=0}^{l-1} \sum_{j=0}^{l-1} z_{i,j} 2^{i+j} = k$$
$$z_{i,j} - x_i - y_j \geq -1 \quad i,j = 0 \ldots l-1$$
$$-2z_{i,j} + x_i + y_j \geq 0 \quad i,j = 0 \ldots l-1$$

Firstly we try to avoid trivial solution and then we equal a binary representation of the number to factorize. Then we represent the partial products to perform factorization. Both first equations avoid trivial solution, meanwhile the third equation gives a binary representation of the number to factorize. Finally, partial products are represented in order to perform factorization.

It is worth to note that in this case of the pseudo-Boolean we encode the direct problem of factorization in contraposition of previously defined encodings, where our effort was devoted to encode the reverse problem of multiplication to solver factorization. Our main goal with this encoding is, given the current state of pseudo-Boolean solving techniques, compare how SAT solvers behave in both cases.

Quadratic Sieve

In order to compare our multipliers with other methods, a good benchmark is to factorize large integers. To do so, we will use Quadratic Sieve with the mathematical software SAGE to compare with our SAT-based multipliers.

Quadratic Sieve (QS) is known as one of the best methods to factorize integers, along with Number Field Sieve, being the best algorithm for integers up to 100 bits long.

The basics of QS are inspired on Fermat's factorization method, trying to find two numbers x and y such that $x \not\equiv \pm y \pmod n$ and $x^2 \equiv y^2 \pmod n$. This means that $(x-y)(x+y) \equiv 0 \mod n$, and we only need to check that $(x-y, x)$ is a non trivial division. As detailed in [Pom85] there is at least a 1/2 chance that the factor will be non trivial. The steps in doing so are the defined, firstly QS defines $Q(x)$ as:

$$Q(x) = (x + \lfloor\sqrt{n}\rfloor)^2 - n = \tilde{x}^2 - n,$$

and compute $Q(x_1), Q(x_2), \ldots, Q(x_k)$. From the evaluations of $Q(x)$, we want to pick a subset such that $Q(x_{i_1})Q(x_{i_2})\ldots Q(x_{i_r})$ is a square, y^2. Then note that for all $x, Q(x) \equiv \tilde{x}^2 \mod n$. At this point,

$$Q(x_{i_1})Q(x_{i_2})\ldots Q(x_{i_r}) \equiv (x_{i_1}x_{i_2}\ldots x_{i_r})^2 \mod n.$$

If conditions above hold, we have factors for n.

2.2.2. Discrete Logarithm

We can define logarithm $log_a(h)$ operation as the solution to the equation $a^x = h$ over the real or complex numbers. Discrete Logarithm is analogue to the logarithm defined over the reals, but now defined over a finite cyclic group instead. So we can say that discrete logarithm is the solution x to the equation $a^x = h$ where a and h are elements of the finite cyclic group G.

Discrete Logarithm is a key concept for several cryptographic procedures such as the standard for digital signature DSS [GFD09], that defines digital signature schemes based on Discrete Logarithm over integer modular arithmetic as well as over elliptic curves. For the first case, the standard specifies private key lengths –roughly speaking x– between 160 and 256 bit length.

All the known methods to tackle the Discrete Logarithm run in exponential time. The naivest approach is based on trial multiplication, encoded for SAT as explained in next section. Other methods performs faster, as Pollard's rho, detailed in Section 2.2.2

Exponentiation

Let's assume two natural numbers a and x to the computed as a^x. Expressing x as binary, $x = \sum_{i=0}^{l-1} x_i 2^i$, we obtain,

$$a^x = a^{2^0x_0+2^1x_1+2^2x_2+\ldots+2^{l-1}x_{l-1}} = \prod_{i=0}^{l-1} a^{2^i x_i}.$$

As x_i are binary variables, only those elements a^{2^i}, with x_i set to one will be multiplied.

Making a sharper look into that technique, we can see that $a^{2^i} = (a^{2^{i-1}})^2$, which means that having a we can easily calculate next series's value squaring the previous one. We can see a modular scheme for this exponentiation method in Fig. 2.2 for an exponent l-bit length.

An easy extension to $\mathbb{Z}_n$ of that fast exponentiation technique may be implemented using modulo n multipliers, with the advantage that any partial product will have length at most $2 \cdot \log n$, so the size of the exponentiation circuit for $\mathbb{Z}_n$ will remain polinominaly bounded in the size of the input.

In Fig. 2.3 there is a modular scheme for a n-modulo multiplier. Once again we use multipliers and adders explained before plus a two's complement module. What we want to encode is $x \cdot y = z \pmod n$, being x and y the multiplicands, n the modulus, and $x \cdot y - k \cdot n$ the output for any

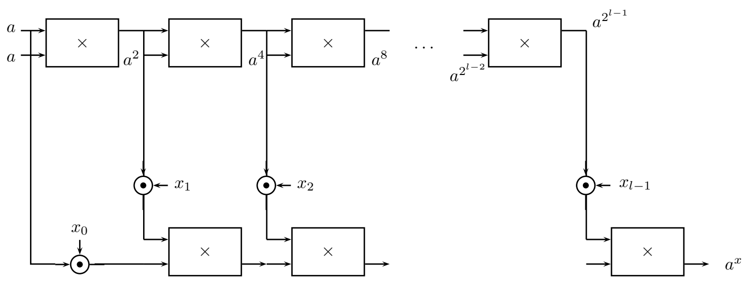


Figure 2.2: Modular design of an exponentiation circuit using multipliers and 1-bit exponentiators.

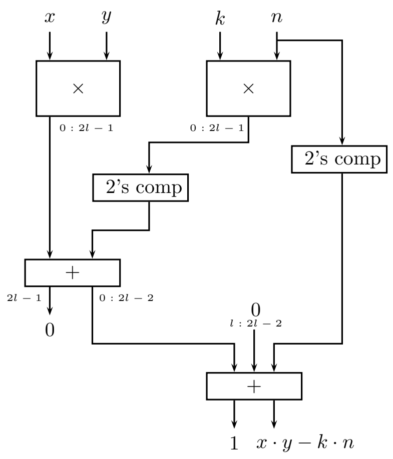


Figure 2.3: Modular design of a n-modulo multiplier.

$k \in \mathbb{Z}$. Two's-complement arithmetic is used to represent negative numbers. For ease of understanding, next we show Boolean equations for the two's complement module.

$$\bigwedge_{i=0}^{l-1} B_i \leftrightarrow \bar{A}_i$$
$$\bigwedge_{i=2}^{l-1} (C_i \vee B_{i-1} \vee C_{i-1}) \wedge (\bar{C}_i \vee B_{i-1}) \wedge (\bar{C}_i \vee C_{i-1}) \wedge (C_i \vee \bar{B}_{i-1} \vee \bar{C}_{i-1})$$
$$(\bar{C}_1 \vee B_0 \vee C_0) \wedge (C_1 \vee \bar{B}_0 \vee \bar{C}_0) \wedge (C_1 \vee \bar{B}_0) \wedge (C_1 \vee \bar{C}_0)$$

We have A as an input vector, B an auxiliary vector containing the vector A negated, and finally C contains the output. All that vectors are l-bit length.

Pollard's Rho

The Pollard's rho randomized algorithm for the discrete logarithm problem [Pol78], is based on finding a collision on a sequence of integers of the cyclic group $\mathbb{Z}_n$. Such a collision will be defined by two integers Y_i and Y_j of the iteration sequence $\{Y_0, Y_1, \ldots, Y_i, \ldots, Y_j\}$ generated with an iteration function $f : \mathbb{Z}_n \to \mathbb{Z}_n$ and such that $Y_i \equiv Y_j \ (mod\ n)$, where each Y_k of the sequence has the form: $Y_k = a^{k_1} h^{k_2}$. Then, once such a collision is found, depending on the particular exponents of the two matching elements of the sequence, the solution to $a^x = h \ (mod\ n)$ may be obtained.

As with the quadratic sieve algorithm, we have used the implementation of this algorithm available in SAGE.

2.2.3. Experimental Analysis

In order to conduct our experimental investigation we have developed two generators of random problem instances for factorization and discrete logarithm problems. For the factorization problem, we first search two primes of length $n/2$ and we took its product as the number to factorize. For the discrete logarithm, we first search a strong prime q using Gordon's algorithm [MVO96], and then we search a generator g for the cyclic group and randomly select an element h from the group. So, we have that $g^x = h$ (mod q) is our discrete logarithm problem instance.

We have used two solving techniques for the experimental analysis:

- SAT solver: Precosat (v.236)[Bie]. Winner at the SAT Competition 2009 for the application category. We have used this algorithm for solving all the propositional encodings based on Boolean circuits for the problems considered.

- Pseudo-Boolean Solver: Minisat+ [ES06]. We have used this algorithm for solving the pseudo-Boolean encoding of the factorization problems. It uses three main approaches to generate circuits in order to translate them to CNF pseudo-Boolean linear problems, which then can be solved by a SAT-Solver. Those three approaches are:

    - Convert a PB-linear constraint to a BDD.
    - Convert a PB-linear constraint into a network of adders.
    - Convert a PB-linear constraint into a network of sorters.

  See [ES06] for more details about the above encodings for PB-linear constraints.

Our experiments have been run on machines with the following specifications: Rocks Cluster 5.2 Linux 2.6.18 Operating System, AMD Opteron 248 Processor clocked at 1.6 GHz, 1.0GB Memory, and GCC 4.1.2 Compiler.

Our SAT encoding for factorization (labeled as `Precosat Array` and `Precosat Booth`) are compared with PB-encodings (labeled as `Minisat+ Adders`, `Minisat+ BDD` and `Minisat+ Sorters`) as shown in Figure 2.4.

SAT encodings for factorization use any of the propositional encodings for integer multiplication. The output is fixed to the desired value n, and the required additional constraints are added in order to make x and y factors non-trivial.

For illustration, these additional constraints are the first two constraints in the pseudo-Boolean encoding for factorization. It can be noted from Figure 2.4 that our encoding using the Array multiplier performs the best for factorization. Also we can see how the conversion of PB-encoding into a network of adders is performing quite well. Both approaches perform similarly because all of them employ networks of adders. Being so, the Array multiplier will be the selected schema to compare SAT-encoding performance against other benchmarks.

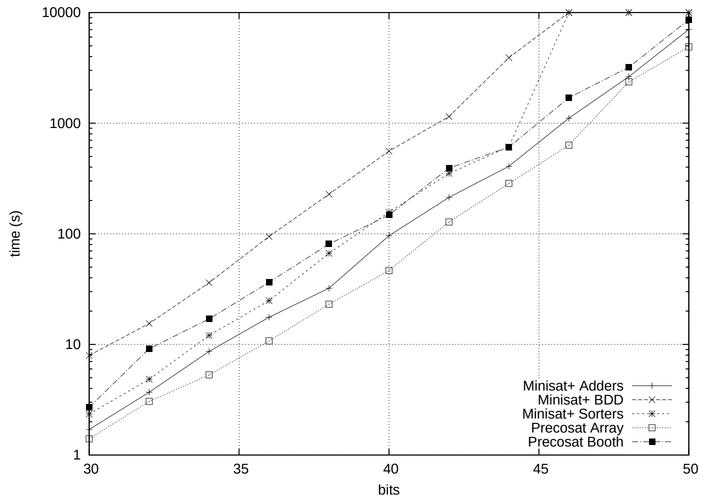


Figure 2.4: Results on Factorization.

Figure 2.5 shows a comparison between the Array Multiplier SAT encoding and SAGE Quadratic Sieve implementation. We can see that our SAT approach performs significantly worse. The results show that with similar time bounds, Quadratic Sieve is able to factor integers of around 200 bits meanwhile the SAT encoding only factors integers of around 40 bits. The figure also shows a similar comparison between our SAT approach for solving the discrete logarithm and Pollard's Rho algorithm. This time, for a same time bound, the difference in the size of the discrete logarithm problems solved is about 4 times bigger for the specialized algorithm. To check the differences between our approach and the dedicated algorithm, we performed a linear regression in order to see the scaling cost of the specialized algorithms and the SAT-based approaches. We have a cost of $e^{0.4092 \cdot bits}$ for Precosat factorization and $e^{0.06766 \cdot bits}$ for SAGE Quadratic Sieve. And $e^{0.8466 \cdot bits}$ for Precosat discrete logarithm and $e^{0.3865 \cdot bits}$ for SAGE Pollard's Rho. So, these results indicate that there is still a lot of space for possible improvements of SAT encodings for these non-linear arithmetic problems.

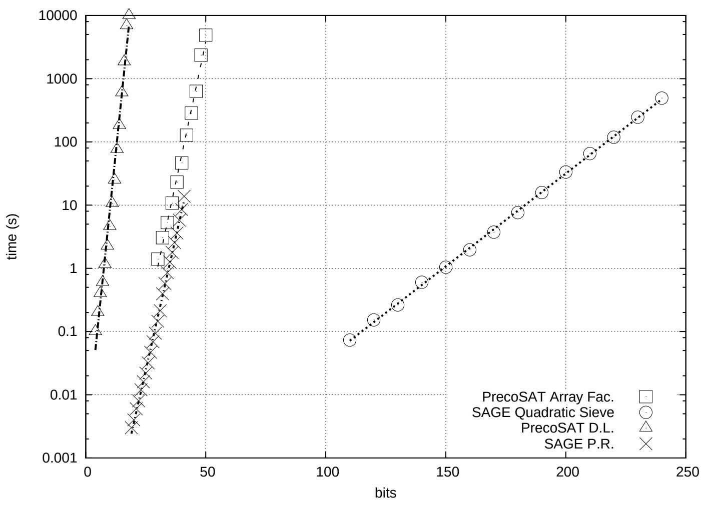


Figure 2.5: Results on Factorization and Discrete Logarithm.

As a final remark, we can say that we have presented SAT encodings of basic non-linear arithmetic operations: multiplication and modular exponentiation, used as the basic building blocks for encoding integer factorization and discrete logarithm as very challenging SAT instances. Our comparison of the performance of our SAT-based methods with the current best performing specialized algorithms has shown that these problems are interesting challenges for the SAT research community, and we honestly think that they deserve further study in order to understand the limits on the performance of SAT encodings for such basic problems.

In the future, we expect to use our SAT encodings for encoding Elliptic Curve problems, where the lack of specialized algorithms for them may make worth the study of the performance of SAT-based algorithms.

2.3. Modular Constraints

Modular arithmetic is used in cryptographic systems as well as in pseudo-random generators, being currently implemented in a wide range of devices, and subject of great interest in media since some evidences shown that pseudo-random generators may have been flawed in order to provide feasible backdoors to certain governmental agencies. Efficient implementations of modular arithmetic are needed for power saving and many environmental constraints requirements [GvHSS07, GW97, NO06, CG93].

Our interest is focused in studying modularity constraints of the following form: $\sum_i c_i \cdot l_i \equiv r \pmod{m}$ where c_i, r and m are positive integers and l_i is a literal of Boolean variables. We will refer to this constraint as a *pseudo-Boolean Modular (PBMod) constraint*. In particular, in our work we are interested in solving, efficiently, formulas which are conjunctions of PBMod constraints.

A straightforward method to deal with PBMod Constraints is through their translation to PB constraints of the form: $\sum_i c_i \cdot l_i \rhd k$ where k is an integer positive constant and $\rhd$ is one of the operators of $\{=, <, \leq, >, \geq\}$. Then, it is possible to use a PB solver with an empty objective function to solve the problem. Some PB solvers also employ, as their solving strategy, the translation to the SAT problem and the usage of SAT solvers. There are several works on the translation of PB constraints to SAT [ES06, BBR09, RM09, BBR06] and cardinality constraints (PB constraints where all coefficients are equal to 1) to SAT [BB03, Sin05, ANORC09].

In [ES06] several encoding schemes into the SAT problem are analyzed: networks of adders, Binary Decision Diagrams (BDD), and networks of sorters. As we show in the experimental analysis that we have carried out, for the problem we address in this work, the encodings based on a network of sorters are the best performing ones. It is well known that an eager transformation may lose some desirable properties/information from the original model. Therefore, we introduce in this work a specialized translation for the PBMod constraints into the SAT problem extending the encoding based on networks of sorters.

We have also compared this approach with other standard solving techniques such as the software package GLPK (GNU Linear Programming Kit): `https://www.gnu.org/software/glpk/`) for solving our Integer Linear Programming (ILP) formulations of the problem.

In order to conduct our experimental investigation we have developed a generator of random PBMod constraints. As in many problems we indeed observe the existence of a phase transition region, i.e. a region where there is an abrupt descent on the ratio of satisfiable instances. We have also studied how the different generator parameters such as: the values of the remainder, the modulo, the coefficients of the variables, and the length of the modular constraints impact on the hardness of the instances and the performance of the different solving strategies. Such study allows to select the appropriate solving technique depending on the structure of the problem.

2.3.1. Modular Constraints as pseudo-Boolean constraints

A modularity constraint on Boolean variables, that we will call *pseudo-Boolean modularity constraint*, is defined by:

$$\sum_i c_i \cdot l_i \ \equiv r \pmod{m} \tag{2.1}$$

where the coefficients c_i and the remainder r are integer constants, the modulo m is a positive integer constant, and the l_i symbols are literals on Boolean variables.
Without loss of generality, this constraint can be rewritten to use just positive integer constants through its normalization. This is achieved by replacing the coefficients and the remainder by their remainder modulo m.

Now, we can naturally translate Eq. 2.1 to a linear integer arithmetic constraint as follows:

$$\sum_i c_i \cdot l_i - k \cdot m = r \tag{2.2}$$

where k is a positive integer variable. We will refer to the translation model of Eq. 2.2 as ModLIA.

Notice that k is actually bounded, i.e., $k \leq \lfloor \frac{C}{m} \rfloor$, where $C = \sum_i c_i$. As we will see in Section 5, in order to solve a problem involving linear integer arithmetic constraints we can use, among other several approaches, a SMT solver with support for the Quantifier Free Linear Integer Arithmetic (QF_LIA) theory or a package like GLPK for Mixed Integer Programming. However, since our modularity constraint is defined on Boolean variables, we may rather be interested on translating Eq. 2.2 into PB constraints. In particular, we just need to express $k \cdot m$ as an arithmetic expression involving only Boolean variables as follows:

$$\sum_i c_i \cdot l_i - \sum_j k_j \cdot m = r \tag{2.3}$$

where k_j are Boolean variables and $j \in \{1, \ldots, \lfloor \frac{C}{m} \rfloor\}$.
We will refer to the translation model of Eq. 2.3 as ModPB-A.

We can still consider a slight variation that can potentially reduce the search space, by reducing the natural symmetry in Eq. 2.3. Notice that the sum of the k_j variables can be the same under different assignments. We can avoid this problem as follows:

$$\sum_i c_i \cdot l_i - \sum_j k_j \cdot j \cdot m = r \ \wedge \ \sum_j k_j \leq 1 \tag{2.4}$$

We will refer to the translation model of Eq. 2.4 as ModPB-B.

As we can see, any consistent assignment to the k_j variables will set at most one variable to true. For both Eq. 2.3 and Eq. 2.4 we can now also apply a PB solver.

2.3.2. Modular Constraints as SAT formulas

Eq. 2.5 is an example of normalized PBMod constraint.

$$3 \cdot b_1 + 2 \cdot b_2 + 5 \cdot b_3 + 3 \cdot b_4 = 1 \pmod 6 \tag{2.5}$$

As we can observe, $C = \sum_i c_i = 13$ and $\lfloor \frac{C}{m} \rfloor = \lfloor \frac{13}{6} \rfloor = 2$.

Then, taking into account the different translation models described in the previous section, we obtain the following PB constraints:

- ModPB-A:

$$3 \cdot b_1 + 2 \cdot b_2 + 5 \cdot b_3 + 3 \cdot b_4 - (k_1 \cdot 6 + k_2 \cdot 6) = 1$$

- ModPB-B:

$$3 \cdot b_1 + 2 \cdot b_2 + 5 \cdot b_3 + 3 \cdot b_4 - (k_1 \cdot 6 + k_2 \cdot 12) = 1 \wedge (k_1 + k_2 \leq 1)$$

  As we can observe, with the ModPB-A model there are two possible ways to sum up to 6 with the k variables, while for the ModPB-B there is only one. In general, for n k-variables, the ModPB-A model considers 2^n consistent assignments while ModPB-B just considers n, thanks to the at-most-one constraint on the k-variables.

  These PB constraints can now be solved with a PB solver. From the different solving techniques of the state-of-the-art solvers, now, we focus on the one that translates the PB constraints to a SAT formula [ES06]. There exist several translation techniques [ES06, BBR09, RM09, BBR06]. Whether a translation is suitable or not depends on both the size of the encoding and the level of consistency that can be achieved under a partial assignment to the Boolean variables. In particular, in [ES06] three main approaches to generating a SAT formula from a PB-constraint are studied.

  - Translate the PB constraint into a BDD, which can be treated as a circuit of ITEs (if-then-else gates) and translated into clauses by the Tseitin transformation [Tse68]. This approach guarantees that the resulting encoding is arc-consistent but its size is exponential in the worst case.
  - Translate the PB constraint into a network of adders. The approach used in [ES06] is similar to the one is used for Data Multipliers to sum up the partial products [Dad68]. The size of the translation is in $O(n)$, however the resulting encoding is not arc-consistent.
  - Translate the PB constraint into a network of sorters. A sorter is a circuit of n input gates and n output gates where the k

lowest output gates are set to true and the rest to false if there are exactly k input gates set to true. The size of the translation used in [ES06] is in $O(n \cdot \log^2 n)$, and although it is not yet arc-consistent it is closer than the translation through adders.

We have used the Minisat+ [ES06] solver to convert the PB constraints, representing modularity constraints expressed with ModPB-A and ModPB-B as to SAT formulas using the three different translation techniques provided by Minisat+: networks of adders, BDDs, and networks of sorters. For the size of problems that will be considered in our experimental evaluation, translation through networks of adders has shown a poor performance (some orders of magnitude worse), BDDs do better but are not as good as networks of sorters. This last translation technique together with ModPB-B is the best performing approach so far. Consequently, in our experimental evaluation the other two translation techniques, adders and BDDs, will not be shown.

Therefore, we have decided to explore whether we can achieve a better encoding if we translate directly the PBMod constraint to a SAT formula based on a network of sorters, instead of having an intermediate state where the PBMod constraint is translated to a PB constraint. In our translation we used the OddEvenMerge sorters [Bat68]. For a more detailed explanation for OddEven mergesort see [Lan00]. In our work the function compare(i_1,i_2) on two input gates (Boolean variables) is translated to a SAT formula, with two new auxiliary Boolean variables o_1, o_2 (the output gates), representing $(o_2 \leftrightarrow (i_1 \wedge i_2)) \wedge (o_1 \leftrightarrow (i_1 \vee i_2))$ with the following clauses:

$$\begin{aligned} o_2 \vee \neg i_1 \vee \neg i_2 \\ \neg o_2 \vee i_1 \\ \neg o_2 \vee i_2 \\ \neg o_1 \vee i_1 \vee i_2 \\ o_1 \vee \neg i_1 \\ o_1 \vee \neg i_2 \end{aligned}$$

A straightforward approach to convert a PBMod constraint into a SAT formula consists in flattening the left hand side (LHS for short) and using a sorter with as many inputs as the sum of the coefficients. Taking into account our example in Eq. 2.5, once flattened,

$$\begin{aligned} & \overbrace{b_1 + b_1 + b_1}^{3\cdot b_1} + \overbrace{b_2 + b_2}^{2\cdot b_2} + \\ \overbrace{b_3 + b_3 + b_3 + b_3 + b_3}^{5\cdot b_3} + \overbrace{b_4 + b_4 + b_4}^{3\cdot b_4} &= 1 \pmod 6 \end{aligned}$$

the sorter would have 13 inputs and we should add the circuit to check if the last output gate activated represents a number congruent to 1 modulo 6. Obviously, this approach is not good in terms of the size of the resulting encoding. So, a better approach consists in connecting several sorters; i.e., creating a network of sorters. We will refer to this encoding as ModCS. In order to apply the encoding based on sorters, the numbers must be represented in unary instead of binary. The sorters play then the role of adders of unary numbers. Notice that the unary representation allows the use of any base for the coefficients. The first phase of the encoding process is to find a base such that the sum of all the digits of the coefficients written in that base is as small as possible, since we want to minimize the entries to the sorters to get the lowest possible size of the encoding. Since this is already a hard optimization problem, we use a brute-force search trying all prime numbers less than 20, as it is done in Minisat+.

In our working example the base we use is $< 1, 3 >$. The coefficients in this base are represented as follows:

$$\begin{aligned} 2 &= (2,0)_{<\mathbf{1},\mathbf{3}>} = 2 \cdot \mathbf{1} + 0 \cdot \mathbf{3} \\ 3 &= (1,3)_{<\mathbf{1},\mathbf{3}>} = 1 \cdot \mathbf{1} + 3 \cdot \mathbf{3} \\ 5 &= (2,1)_{<\mathbf{1},\mathbf{3}>} = 2 \cdot \mathbf{1} + 1 \cdot \mathbf{3} \end{aligned}$$

As we can see in Fig. 2.6, two sorters are used in our example. One for the contribution of every coefficient to the 1-bits weight and another for the 3-bits weight. Notice that one of the input gates of the 3-bits sorter is connected to the third output gate of the 1-bits sorter. This is the way we represent the carry between sorters. Then, taking into account that the network of the sorters represents the sum of the LHS, we just need to add the additional circuitry, i.e. the comparator, that checks whether $LHS \in \{1, 7, 13\}$, the possible numbers congruent to 1 mod 6. For example, in order to assure that $LHS = 1$ we just need to check if the first output gate of the sorter 1-bits is true, the second is false and if the first output gate of the sorter 3-bits is also false. Notice that if the k_{th} output gate of a sorter is false then the rest of the upper gates must also be false. This circuitry can now be reused to test the conditions $LHS = 7$ and $LHS = 13$.

The motivation of this approach is to reduce the size of the encoding as much as possible, both in terms of clauses and variables while preserving the good propagation properties. The PB solver Minisat+ uses in its translation of the PB constraints to a CNF formula an intermediate circuit representation where any two syntactically identical nodes are merged by the so-called structural hashing. This reduces effectively the size of the encoding, however our approach still generates smaller formulas. Table 2.3.2 compares for two sets of the experimental results the size of our encoding approach, ModCS, to the conversion through Minisat+ and the option of sorters on the model ModPB-B.

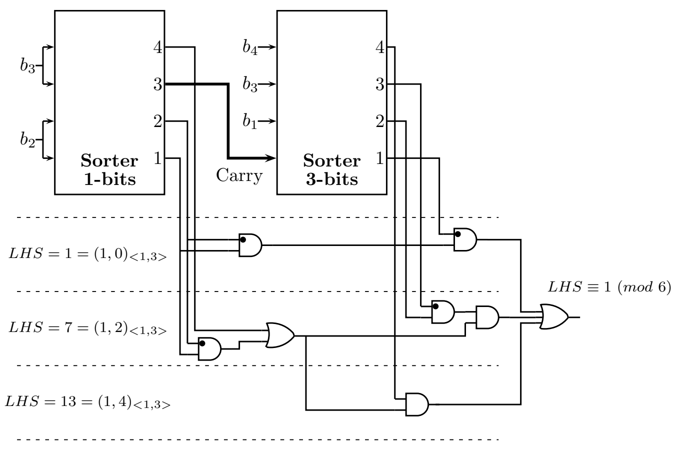


Figure 2.6: Schema for the ModCS encoding applied to example of Eq. 2.5.

(v,n,M,f)	Variables	Clauses	Vars. × Clauses
100,10,5,56			
ModPB-B	9,089	25,414	$2.3 \cdot 10^8$
ModCS	7,203	18,258	$1.31 \cdot 10^8$
		Ratio	1.75
100,50,5,22			
ModPB-B	60,988	179,466	$1.09 \cdot 10^{10}$
ModCS	33,249	89,880	$2.98 \cdot 10^9$
		Ratio	3.66

Table 2.1: Comparison of the size of encodings of models ModPB-B (Minisat+ and option sorters) and ModCS.

2.3.3. Experimental Analysis

In order to conduct our experimental investigation we have developed a generator of random pseudo-Boolean Modularity constraints, *randPBMod(v,n,M,f)*. The generator creates a formula consisting on f PBMod constraints, each one with n Boolean variables out of a total of v Boolean variables. The constraints are already normalized according to the respective modulo. For each PBMod constraint, the modulo m, the remainder r, the coefficients c_i and the variables b_i are selected from a uniform random distribution in the intervals $m \in [2, M]$, $r \in [0, M-1]$, $c_i \in [1, M-1]$ and $b_i \in [1, v]$. The variables are selected with no repetition for every PBMod constraint.

We have used several solving techniques for the experimental analysis:

- SAT solver: Precosat (v.236). Winner at the SAT Competition 2009 for the application category.

- PB solver: Bsolo (v.3.1) [MMS05]. Winner at the Pseudo-Boolean Evaluation 2009 for the category "no optimization, small integers, linear constraints" in sat+unsat answers.

- GLPK: advanced ILP solver with cutting planes. We have used the advanced branch and bound GLPK ILP solver, that preprocesses the input ILP problem for obtaining a simplified version of it, and we have also used the option of adding cover, clique and Gomory cutting planes, to further improve the pruning of the search space. For a general introduction to cutting planes, see for example [MMWW02].

In this section we show results for each of these solvers (labeled as `psat`, `bsolo`, and `gplk`) using each one at least one of the following encodings: ModPB-A, ModPB-B, ModCS, and ModLIA (§ Section 2.3.2). Take into account that the encodings ModPB-A and ModPB-B, in order to be used by a SAT solver, should be first translated into SAT with Minisat+ using the sorting networks option.

Our experiments have been run on machines with the following specifications: Rocks Cluster 5.2 Linux 2.6.18 Operating System, AMD Opteron 248 Processor clocked at 1.6 GHz, 1.0GB Memory, and GCC 4.1.2 Compiler.

In order to evaluate the different encoding performances we have run our experiments for a wide range of parameters. The number of PBMod constraints in a formula (f) determines its satisfiability. As plotted in Fig. 2.7, the probability of having an unsatisfiable problem grows with f, as it becomes more constrained. In this case, when problems have $v \simeq n$, the satisfiability transition occurs at low values of f and there seems to be a sharp transition, like the one observed in other NP-complete problems, as SAT [SK96, AP04] or CSP [XL00]. By relaxing this condition and allowing values of v larger than n, we can build satisfiable problems for larger values of f.

As a first benchmark, we have designed a set of experiments for $v = 100$, $n = 10, 20, 50$ and $M = 5, 10, 20$, testing values for f up to a given time out and solving 100 independent instances per point. This benchmark shows a first picture about the problem hardness parameter dependency, and proves that the problems become harder as n, M and f grow. It also helps to look at the performance differences of our encodings in distinct benchmark conditions.

Figure 2.8 shows the ratio of performance, considering solving time using Precosat, between the two best performing encodings for SAT solvers: ModCS and ModPB-B converted to SAT with Minisat+ using the sorting networks option. Measures are done for the values of v, n (legend), and M (y-axis) detailed above. The boxes cover the region of the number of PBMod constraints (f, x-axis) where both encodings solve problems in the

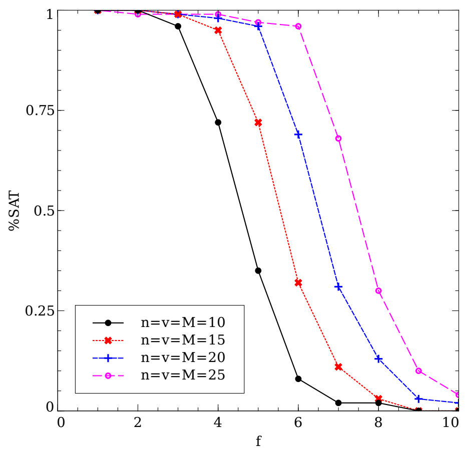


Figure 2.7: Percentage of satisfiable instances as as function of f.

allotted time (the time out used has been 30 minutes), and still give significant times, i.e. more than 1 second. Inside each box figures represent the performance ratio of ModCS over ModPB-B, it is measured as the mean of the performance rations over the considered range of f. As an example, a ratio of 1.77 means that ModCS is 1.77 times faster than ModPB-B.

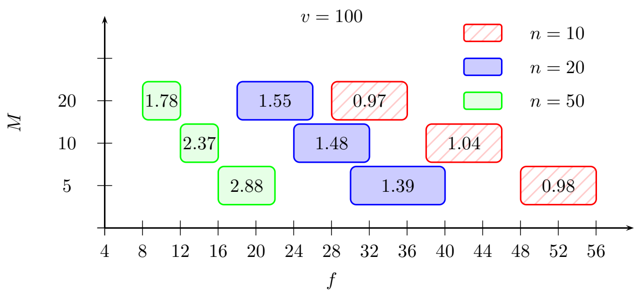


Figure 2.8: Performance ratio for the two best encodings -ModCS and ModPB-B-, using Precosat v.236 for $v = 100$ and distinct values for n and f.

A first conclusion from Fig. 2.8 is that ModCS performs better as the number of variables per constraint (n) increases. It should be noted that as n increases, so it does the sum of the coefficients (C), leading to more distinct interpretations of a congruence for a given M, and finally, giving more chances of reusing circuitry to ModCS encoder (as explained in the previous section), reducing the formula size. Second, as a result of the same

	f	ModCS	ModPB-B	ModPB-Bbin	Ratio
$M = 5$	36	912	1193	–	1.31
	34	158	297	765	1.88
	32	33	82	169	2.48
	30	8.2	32	63	3.90
	28	3.6	16	14	3.89
	26	1.9	9.3	6.1	3.21
	24	1.6	4.8	2.5	1.56
				Mean Ratio	**2.17**
$M = 20$	22	307	371	442	1.21
	20	19	45	32	1.68
	18	3	16	5.3	1.77
	16	2	7.4	2.3	1.15
	14	1.6	4.6	1.7	1.06
				Mean Ratio	**1.37**

Table 2.2: Median time for ModCS, ModPB-B and ModPB-Bbin encodings using Precosat v.236. $v = 200$, $n = 50$ and their relative performance (– means median time larger than 30 minutes).

effect, one can also observe, for $n = 50$, an increase of relative performance for ModCS encoder as M decreases, due to the larger number of constraints (f).

To prove the scaling with the number of variables, Table 2.3.3 reports the performance ratios for $v = 200$, $n = 50$ and $M = 5, 20$. The mean ratio between encodings raises up to 3.13 and 3.1 respectively.

Tables 2.3.3, 2.4, 2.5, and 2.6 show the median time to solve all the instances and the percentage of solved instances for a timeout of 30 minutes, and a wide range of parameter values. Left column denotes the employed combination solver/encoding as mentioned at the beginning of the current section.

Table 2.3.3 shows that ModPB-A is not a good option for any solver. It also shows that for solvers `yices` and `glpk`, none of both PB models (ModPB-A, ModPB-B) is competitive with ModLIA.

On the one side, it also seems clear that the LP (gplk/ModLIA) approach is not competitive with the other best combination encoding/solver.

On the other side, the differences between the SAT and PB solvers are more subtle. With respect to both encodings used for Precosat solver, as mentioned above, our improvement of the translation through network of sorters (ModCS) slightly outperforms the original translation (ModPB-B) in most cases, increasing their differences as the number of variables (n) be-

	f				
	6	**7**	**8**	**9**	**10**
bsolo/ModPB-A	−/26	−/8	−/3	−/0	−/1
bsolo/ModPB-B	647/53	−/18	−/9	−/3	−/3
psat/ModPB-B	4.1/100	12/100	13/100	12/100	12/100
psat/ModCS	4.3/100	14/100	16/100	16/100	17/100
glpk/ModPB-A	−/3	−/1	−/0	−/0	−/0
glpk/ModPB-B	55/97	130/96	245/95	354/97	373/95
glpk/ModLIA	12/97	25/97	28/95	25/97	24/97

Table 2.3: Median solving time / % of solved instances within alloted time for v,n and M=20 (− means median time larger than 30').

comes larger, and particularly for low values of M, that is when the ModCS encoding ameliorates its circuit reusing capabilities. According to the PB solver performance (bsolo/ModPB-B), it does particularly well for underconstrained problems, i.e. when f is low, but tends to solve less instances than psat/ModCS for higher f. In order to reinforce this last point, we have conducted additional experiments for $v = 200$ and $n = 50$ as shown in Table 2.3.3.

As a final remark we can say that we have studied how to solve efficiently PBMod Modularity constraints. Although these constraints are naturally expressed as Linear Integer Arithmetic constraints, we have shown that is not the best approach at least when the LHS term of the PBMod involves only Boolean variables. We have proposed two possible alternative approaches: (i) a translation to PB constraints (models ModPB-A and ModPB-B) and the usage of *pure* PB solver or a PB solver based on a translation into SAT, and (ii) a direct translation into a SAT formula based on a network of sorters (model ModCS) and the usage of a SAT solver. For the first approach we have provided a new encoding (ModPB-B) that is more competitive than the naive conversion to PB constraints (ModPB-A) by breaking the symmetry in that encoding. However, the second approach (ModCS) is even more competitive since it allows to better exploit the expressiveness of the original model by producing smaller encodings while preserving the propagation properties.

It remains for future work to test other types or variations of sorter schemes, such as the *Half* sorting networks based on OddEven *Half* Merging networks, see [ANORC09], where the main contribution are the Cardinality Networks based on Simplified Merging Networks. This approach saves a lot of circuitry when the RHS term is small enough compared to the number of the inputs of the sorter. However, in our current approach, we cannot benefit from those savings as even when we have a small remainder term

Table 2.4: Median solving time / % of solved instances within alloted time for v=100 and n=10 (– means median time larger than 30')

	f				
M=5	**48**	**50**	**52**	**54**	**56**
bsolo/ModPB-B	1.2/99	11/94	107/71	–/36	–/17
psat/ModPB-B	1.1/100	2.5/100	9/99	42/95	286/50
psat/ModPB-Bbin	1.1/100	3/100	13/99	52/99	298/75
psat/ModCS	0.8/100	3.3/100	8.3/95	53/95	299/75
glpk/ModLIA	–/0	–/0	–/0	–/0	–/0
M=10	**38**	**40**	**42**	**44**	**46**
bsolo/ModPB-B	6.8/81	–/47	–/21	–/9	–/2
psat/ModPB-B	2.9/100	13/97	130/88	720/71	1391/55
psat/ModPB-Bbin	2.9/100	14/96	131/90	643/72	1299/54
psat/ModCS	2.3/100	15/98	104/83	748/67	1598/51
glpk/ModLIA	–/0	–/0	–/0	–/0	–/0
M=20	**28**	**30**	**32**	**34**	**36**
bsolo/ModPB-B	1.3/92	68/74	371/56	292/51	358/67
psat/ModPB-B	0.8/100	1.9/100	2.9/100	3.1/100	3.1/98
psat/ModPB-Bbin	0.6/100	1.7/100	2.6/100	3.0/100	2.4/100
psat/ModCS	0.7/100	1.8/100	3.0/100	3.5/100	3.0/100
glpk/ModLIA	–/0	–/0	–/0	–/0	–/0

we need to consider several output gates in order to test the modularity condition.

2.4. Routing and Wavelength Assignment Problem

Finding challenging benchmarks for SAT solvers is not only interesting from an evaluation of algorithms efficiency point of view, but also interesting in the theoretical computer science community. However, most of the efforts in this area are not devoted to solve real world problems. There are some real world applications of SAT, such as such as Electronic Design Automation (EDA) where SAT solving techniques become a suitable tool for problem instances generated. This section focuses in networking problems, in particular in solving constraints which appear when introducing new optical technologies in backbone infrastructures.

Last years have seen a surge in all-optical network deployment that has

Table 2.5: Median solving time / % of solved instances within alloted time for v=100 and n=20 (− means median time larger than 30')

	f				
M=5	**32**	**34**	**36**	**38**	**40**
bsolo/ModPB-B	2.0/100	16/91	128/69	−/31	−/6
psat/ModPB-B	4.8/100	13/100	62/99	340/87	1550/53
psat/ModPB-Bbin	5.9/100	14/100	73/100	393/86	−/45
psat/ModCS	3.1/100	11/100	38/100	312/89	1064/60
glpk/ModLIA	−/0	−/0	−/0	−/0	−/0
M=10	**24**	**26**	**28**	**30**	**32**
bsolo/ModPB-B	0.5/100	8.3/95	−/48	−/12	−/5
psat/ModPB-B	2.7/100	11/99	99/92	1427/53	−/20
psat/ModPB-Bbin	1.9/100	11/99	80/96	885/61	−/20
psat/ModCS	1.1/100	9.5/100	85/87	1239/56	−/21
glpk/ModLIA	−/0	−/0	−/0	−/0	−/0
M=20	**18**	**20**	**22**	**24**	**26**
bsolo/ModPB-B	0.1/100	1.4/95	1175/47	−/15	−/8
psat/ModPB-B	1.4/100	5.0/100	50/96	1423/55	−/10
psat/ModPB-bin	0.8/100	3.1/100	32/94	1033/55	−/13
psat/ModCS	0.7/100	2.9/100	35/95	1262/58	−/12
glpk/ModLIA	−/0	−/0	−/0	−/0	−/0

come together with a dramatic increase in available bandwidth thanks to the use of *Wavelength Division Multiplexing* (WDM for short) on such networks.

These kind of networks work by allocating direct connections (circuits) between users, usually customers, traversing all the network. As the main idea is to use all-optical networks, such circuits must provide light continuity, and given that possible paths or routes are a finite resource, and that for every connection between network nodes only a limited set of *wavelengths* (or lambdas) are available, there is much interest in devising methods and algorithms to efficiently allocate routes and lambdas to each required connection (such pairs, route and lambda, are known as a lightpath).

This problem is known as *Routing and Wavelength Assignment* (RWA for short) problem. It can be solved in three flavours: as a static problem, knowing in advance all traffic demands we must attend (known also as RWA-SLE, *Static Lightpath Establishment*); as an incremental problem, where new demands can appear at any time, but once established they stay so for ever; or as a dynamic problem, where demands appear spaced in time and must be attended as they appear, and after a working period they disappear and

Table 2.6: Median solving time / % of solved instances within alloted time for v=100 and n=50 (− means median time larger than 30')

	f			
M=5	**16**	**18**	**20**	**22**
bsolo/ModPB-B	0.7/100	5.8/99	92/86	−/44
psat/ModPB-B	10/100	27/100	100/100	542/80
psat/ModPB-Bbin	5.9/100	27/100	112/98	723/73
psat/ModCS	1.9/100	10/100	49/100	363/93
glpk/ModLIA	1493/53	−/15	−/3	−/0
M=10	**12**	**14**	**16**	
bsolo/ModPB-B	0.4/100	8.1/89	−/46	
psat/ModPB-B	8.7/100	40/98	379/83	
psat/ModPB-Bbin	3/100	31/100	323/85	
psat/ModCS	2.1/100	24/100	272/83	
glpk/ModLIA	−/13	−/2	−/0	
M=20	**8**	**10**	**12**	
bsolo/ModPB-B	0.1/100	0.5/97	1559/50	
psat/ModPB-B	2.5/100	12/100	94/98	
psat/ModPB-Bbin	1.1/93	6.3/98	165/97	
psat/ModCS	1/100	3.5/100	96/97	
glpk/ModLIA	1341/57	−/3	−/0	

	f			
M=5	**30**	**32**	**34**	**36**
bsolo/ModPB-B	2.9/100	16/98	131/91	1562/52
psat/ModCS	8.2/100	33/100	158/91	912/69
M=20	**18**	**20**	**22**	
bsolo/ModPB-B	0.3/100	3.4/98	189/63	
psat/ModCS	3/100	19/100	307/82	

Table 2.7: Median solving time / % of solved instances within alloted time for v=200 and n=50.

resources can be reclaimed for reuse. RWA has been proved to be NP-complete [CGK92], thus making it an interesting research problem from the optimization point of view.

We propose a new approach to encode the previously mentioned RWA-SLE problem as a pseudo-Boolean satisfiability problem. Our new approach is then compared to a previously published SAT encoding approach [VSK05][1] and to greedy algorithms traditionally used in the networking literature. There is also recent work about the problem of maximizing the number of traffic demands that can be attended with a given network using constraint programming [Sim09]. Experimental results clearly show that our approach outperforms other methods on instances where those methods have poor performance or are unable to provide a solution. Those cases used to be scenarios with a critical level of resources and with none or a few possible solutions.

We also provide an ASP approach to solve the RWA-SLE problem. Answer Set programming has become an attractive tool for representation and reasoning. Although some solvers were proposed in [SNS02] [LPF$^+$06], recent work in ASP solving techniques such as conflict driven learning, backjumping, restart and watched literals lists has been proposed in [GKNS07], making Clasp the best performing ASP solver. This can bee seen in [GLN$^+$07] [DVB$^+$09] [CIR$^+$11] where Clasp and the Potassco framework show very good performance on global results. Moreover, Clasp also showed very good performances in SAT competitions, making ASP a good technique for CSP solving.

We compare this new ASP encoding approach for the RWA-SLE problem with previous works based on SAT and pseudo-Boolean satisfiability techniques shown as best for RWA-SLE decision problem. Experimental results show that ASP approximation can improve the performance under certain circumstances described below.

2.4.1. Definitions

We present some formal definitions for the problem we study in this paper. The routing and wavelength assignment problem (RWA) without wavelength translation is a generic term. In our case we focus on the RWA-SLE without wavelength conversion on nodes. It is worth to take into account that dealing also with wavelength conversion, would be another interesting problem which is RWA-*Translation* (RWA-T for short).

Every instance of the RWA-SLE problem is characterized by the following elements:

- Connection network $N = (V, E)$, where V is the set of nodes and E contains an undirected link $l_{i,j}$ for every pair of nodes i and j such that

[1]We compare with this approach as it seems to be the "best" existing SAT approach.

there is a link between them. To simplify the problem we consider that the capacity, different number of wavelengths a link can support, of all links is the same, denoted by parameter λ.

- Set of traffic demands R, where $r_{s,t} \in R$ if there is a demand for a lightpath between s and t.[2] In RWA without wavelength conversion, the wavelength must be the same for every link corresponding to the same route $r_{s,t}$, while in RWA-T it is assumed that a node can translate a wavelength to any other.[3]

2.4.2. SAT Based Encodings

Once the problem is defined, we describe two main encoding approaches. First, we encode RWA using pseudo-Boolean formulation (PB). A first pseudo-Boolean encoding is described in [ARA07]. We extend it by adding more powerful clauses and by proposing a more compact formulation that performs better for large problems. Second, we use a SAT formulation proposed in [VSK05], applying it to a broader range of problems and comparing it with our PB approach.

It is worth noting that a PB encoding allows to easily extend the formulation to the RWA-T problem, as for example, translating the linear programming approach of [TK08] to PB. Such an extension in SAT is a much harder task. Note that the solution for RWA-T in [VSK05] is only valid for full-capable conversion nodes and cannot be generalized to network scenarios where the conversion range of the nodes is limited, as in real world networks.

Pseudo-Boolean Encoding

Our encoding is based on a set of propositional variables and constraints specified with pseudo-Boolean formulas. Some of the pseudo-Boolean constraints have a direct translation to logical clauses, but some of them have a more compact form as the pseudo-Boolean constraints we present.

For every route demand $r_{s,t} \in R$ we have the following propositional variables to encode all the possible routes in the communication network (V, E):

1. For every node $i \in V$, variable $n^i_{r_{s,t}}$ indicates if node i appears in the route for $r_{s,t}$.

[2]If we allow multiple lightpaths between each possible pair of nodes, then R is a multiset instead of a set. This can happen in "real-world" networks, as each lightpath can correspond to a different customer or application.

[3]In fact, usually not all nodes can convert wavelengths, and those that can are able only of a limited range and number of translations.

2. For every link $l_{i,j}$, variable $d^{i,j}_{s,t}$ indicates if link $l_{i,j}$ appears in the route for $r_{s,t}$.

3. For every link $l_{i,j}$ and wavelength w, variable $x^{i,j,w}_{s,t}$ indicates if link $l_{i,j}$ appears in the route for $r_{s,t}$ using wavelength w.

So, the total number of propositional variables is $|R| \cdot (|V| + |E| \cdot (\lambda + 1))$.

The set of constraints can be divided in two groups, one that ensures the selection of a connected path for each route demand $r_{s,t}$, and another that avoids using the same wavelength for routes sharing a link. For the first group, we have the following pseudo-Boolean constraints:

- The starting and ending nodes of the route must be active:

$$n^s_{r_{s,t}} = n^t_{r_{s,t}} = 1, \ \forall r_{s,t} \in R \tag{p1}$$

- On a route, there is only one active link connected with s and t:

$$\sum_{l_{s,i} \in E} d^{s,i}_{s,t} = 1, \ \forall r_{s,t} \in R \tag{p2}$$

$$\sum_{l_{i,t} \in E} d^{i,t}_{s,t} = 1, \ \forall r_{s,t} \in R \tag{p3}$$

- For any node k, different from s and t, either it is active together with two adjacent links or it is not active and none of its links are active:

$$-2n^k_{r_{s,t}} + \sum_{k \neq \{s,t\}, j \neq \{s,t\}} d^{k,j}_{s,t} = 0, \ \forall r_{s,t} \in R \tag{p4}$$

For the second group, we have these constraints:

- If a link $l_{i,j}$ is active, then exactly one wavelength is used in that link, otherwise no wavelength is used:

$$-d^{i,j}_{s,t} + \sum_{w} x^{i,j,w}_{s,t} = 0, \ \forall r_{s,t} \in R, \ \forall l_{i,j} \in E \tag{w1}$$

- For any intermediate node (j) in a route, if a wavelength w is used in link $l_{i,j}$ for a given route, then it must also be used in adjacent link $l_{j,k}$ that is active in the same route:

$$-x^{i,j,w}_{s,t} - d^{j,k}_{s,t} + x^{j,k,w}_{s,t} > -2, \quad \forall r_{s,t} \in R, \quad \forall (l_{i,j}, l_{j,k}) \in E^2 | j \neq s,t; i \neq k \tag{w2}$$

  Observe that assuring the use of the same wavelength between adjacent links of the route is enough to ascertain the use of only on wavelength in all the links of the route.

- The same wavelength w cannot be used in a given link $l_{i,j}$ for different routes:

$$-x_{s,t}^{i,j,w} - x_{s',t'}^{i,j,w} > -2, \ \forall (r_{s,t}, r_{s',t'}) \in R^2 | \{s,t\} \neq \{s',t'\} \quad \forall l_{i,j} \in E, \ \forall w \tag{w3}$$

Related to this second group, two redundant or alternative constraints may be considered.

- For any route, if the starting active link $l_{s,i}$ uses wavelength w, then any other different link $l_{i',j'}$ in the route cannot use a different wavelength w' (and analogously for the ending active link $l_{i,t}$):

$$\begin{aligned} -x_{s,t}^{s,i,w} - x_{s,t}^{i',j',w'} &> -2, \ \forall r_{s,t} \in R, \quad \forall (l_{s,i}, l_{i',j'}) \in E^2, \ \forall w \neq w' \\ -x_{s,t}^{i,t,w} - x_{s,t}^{i',j',w'} &> -2, \ \forall r_{s,t} \in R, \quad \forall (l_{i,t}, l_{i',j'}) \in E^2, \ \forall w \neq w' \end{aligned} \tag{w2b}$$

  This constraint achieves the same effect that w2 because for a connected path it is enough to ensure the use of a same wavelength between adjacent links.

- A wavelength w is used by at most one of the route demands in a link:

$$\sum_{r_{s,t} \in R} x_{s,t}^{i,j,w} \leq 1, \ \forall l_{i,j} \in E, \ \forall w \tag{w3b}$$

  This constraint achieves the same effect as constraint w3.

Given the complementarity between constraints w2 and w2b, and between w3 and w3b, we can consider four different PB encodings: PB_{2+3}, $\mathrm{PB}_{2+3\mathrm{b}}$, $\mathrm{PB}_{2\mathrm{b}+3}$, and $\mathrm{PB}_{2\mathrm{b}+3\mathrm{b}}$ obtained by selecting one from each pair of complementary constraints. It is worth noticing that in a solution of the previous constraints although for any route $r_{s,t}$ a connected path must be present, it can also contain some other activated links that will form isolated cycles. However, these cycles do not change the soundness of the encoded path between s and t.

A More Compact pseudo-Boolean Encoding

The previous PB encoding considers individual variables for encoding the wavelength used in each link of a route, with the idea to be able to extend it to an encoding for the general RWA-T problem, where some nodes may be able to convert wavelengths. In case we are only interested in working with the non conversion variant of RWA, we can reduce the number of variables and constraints as we show in the following modified PB encoding (called compact PB encoding, PB_{cpt}).

The number of variables and constraints is reduced by considering only one variable $x^w_{s,t}$ per traffic demand and wavelength, denoting that the same wavelength w is used in all the links of the route between s and t. This new encoding substitutes the previous second group of constraints (set w) by the following:

- One wavelength must to be used per demand:

$$\sum_w x^w_{s,t} = 1, \ \forall r_{s,t} \in R \tag{cw1}$$

- For each pair of traffic demands sharing a link, at most one wavelength variable must be set:

$$-d^{i,j}_{s,t} - d^{i,j}_{s',t'} - x^w_{s,t} - x^w_{s',t'} > -4, \quad \forall (r_{s,t}, r_{s',t'}) \in R^2 | \{s,t\} \neq \{s',t'\}, \ \forall l_{i,j} \in E, \ \forall w \tag{cw2}$$

Comparison with SAT Encoding

In [VSK05] a SAT encoding for RWA is presented, i.e., based on a set of Boolean variables plus a CNF formula for encoding the problem. They showed that their approach outperformed previous specialized approaches [KO04] for solving RWA. Since we noticed that for some constraints a more compact pseudo-Boolean formulation was possible, and given the state of the art in efficient translations of pseudo-Boolean constraints to CNF formulas, it seemed natural to study possible improvements of their encoding by using a pseudo-Boolean encoding that was more compact and efficient, at least for some cases. That is the main reason for studying the encoding we present in this paper.

For each combination of node, route and wavelength, the SAT encoding of [VSK05] uses a Boolean variable for indicating whether that node is active for that route using that wavelength. Analogously, for each combination of link, route and wavelength, it uses a Boolean variable for indicating whether that link is active. The fact that they do not use a specific variable for encoding whether a link is active for one route, independently of the wavelength used, is one of the reasons that increases the number of clauses of the problem with respect to the number of clauses in our encoding.

For the clauses used for encoding the constraints, we discuss only the ones that present significant differences with our encoding. When we consider the size of our PB constraints, we are assuming that we transform them to a CNF using the sorting network encoding of [ES06]. Recently, new CNF encodings for PB constraints have been proposed [BBR09] that, although they have a bigger size, they promise to be more powerful, with respect to propagation, in some cases. However, they still have to be successfully integrated in current SAT solvers.

To ensure end-to-end continuity for a route, the SAT encoding forces each node of the path to have the right number of active links in a different way. For ensuring that the start and ending nodes of a route have exactly one active link, it uses a set of clauses for ensuring *at least one link* plus a set of clauses for ensuring *at most one link*. This second set of clauses is the one that gives a significant difference in the number of clauses needed, compared with our corresponding PB constraints p2 and p3, as the number of clauses for this *at most one link* is $O((\lambda\ deg)^2)$ for the starting or ending node of a route but for our constraints p2 and p3 is $O(deg \log^2(deg))$. [4] We have an analogous situation for the constraint that ensures that the other nodes of the route have exactly two active links. The SAT encoding uses a set of clauses for ensuring *at most two active links per node*, that has a size of $O(deg^3\lambda^2)$, for each node of the route. By contrast, our corresponding PB constraint p4 has a size of $O(deg \log^2(deg))$ for each node of the route.

For ensuring that the same wavelength is used along all the links of a route, the SAT encoding uses a set of clauses that ensures that if a wavelength is used on the active link connected to the starting node of the route, then the other active links of the route cannot use other wavelengths, and analogously with the ending node. This set of clauses has size $O(|E|\lambda^2 deg)$ for each route demand. In contrast, we get the same effect with our sets of PB constraints w1 and w2 that have size $O(|E|\lambda(\log^2(\lambda) + deg))$. Actually, the set of clauses used by the SAT encoding is equivalent to the set of PB constraints defined in our alternative constraint w2b. For the $\mathrm{PB_{cpt}}$ encoding the only constraint we need is cw1, that has size $O(|R|\lambda)$, smaller than in the other encodings.

To avoid using the same wavelength in a link in more than one route, the SAT encoding uses the set of clauses resulting from the transformation of the constraints defined in w3, so both have the same size: $O(|R|^2\lambda|E|)$. However, given that the alternative PB constraint w3b is a cardinality constraint[5], using the sorting network CNF encoding the total number of clauses will be $O(|R| \log^2(|R|)\lambda|E|)$. So, given that the level of consistency achieved by both w3 and w3b is the same, arc-consistency for the constraint: *a wavelength is used in any link by at most one route*, in the case of problems with a high number of route demands, using w3b instead of w3 can be an advantage. For the $\mathrm{PB_{cpt}}$ encoding, constraint encoding cw2 has a bigger size than w3b: $O(|R|^2\lambda|E|)$. So, even if $\mathrm{PB_{cpt}}$ is more compact with respect to the number of variables, this is done at the cost of increasing the size of the encoding for this constraint.

[4]deg is an upper bound on the number of links of any node.

[5]A pseudo-Boolean constraint in which all the coefficients are 1.

2.4.3. Experimental Analysis for SAT-Based Encodings

To compare the performance of the above explained encodings we conduct experiments in two cases. First, by generating synthetic problems in a broad range of network scenarios by developing a problem generator and creating a set of instances. Second, by translating and encoding existing benchmarks based on real world networks. Although there exist some problem instances derived from those networks [KO04, VSK05], we have found that they are too easy for all the solving approaches we compare in this section, probably because these instances are defined with a very high level of resources.[6]

A RWA Problem Generator

Our generator works by first creating a fully connected network and then generating a set of random traffic demands to be satisfied by the network. Problem instances for our experimentation are created using Waxman model [Wax88], which creates a network topology that obeys a power law. Waxman model works by first placing nodes randomly across a bidimensional space, as is the case with real networks. With nodes placed then it proceeds to add edges (optical connections) between each pair of nodes u and v, using the probability function, $P(u,v) = \beta^{\frac{-d(u,v)}{L\alpha}}$ where $d(u,v)$ is the distance on the plane between u and v, L is the maximum distance between two nodes, and α and β are parameters in the range $(0,1]$. Larger values of β represent higher connectivity degrees for the nodes, whilst α indicates connectivity from a node to more distant nodes, i.e. long-haul edges. For each edge we must also define the number of available wavelengths (λ).

Once the network is built, we generate a set of traffic demands D between random pairs of nodes. As the network is fully connected there is always a path between every possible pair of nodes but, as satisfying a demand uses up an available wavelength, there is no guarantee that all possible sets of demands will be satisfiable.

In order to evaluate the encodings performance, we have used a test set consisting of 980 instances, for network topologies from 10 to 20 nodes, from 20 to 50 traffic demands, and 2 to 10 available lambdas, creating a wide range of problems from unsatisfiable to easily satisfiable.

Hardness Characterization

In RWA problems, as in other studied problems [Pro96, AGKS00, ABF$^+$10], there exists an easy-hard-easy hardness characterization when we move from unsatisfiable to satisfiable instances along a given parameter, the number of

[6]Due to space constraints, detailed results for real world networks are not included, although authors have them available on request.

	Satisfiable					
Time(s.)	2+3	2b+3	2+3b	2b+3b	cpt	SAT
< 1	9	1	32	5	0	14
$(1, 10]$	42	3	161	2	1	124
$(10, 10^2]$	3	2	2	0	17	29
$(10^2, 10^3]$	0	0	0	0	2	2
$\geq 10^4$	0	0	0	0	0	0
$\sum$	54	6	195	7	20	169
	Unsatisfiable					
Time(s.)	2+3	2b+3	2+3b	2b+3b	cpt	SAT
< 1	10	9	55	25	31	19
$(1, 10]$	10	3	180	16	10	15
$(10, 10^2]$	1	1	4	1	5	0
$(10^2, 10^3]$	2	2	2	1	1	0
$\geq 10^4$	0	2	0	1	8	0
$\sum$	23	17	241	44	55	34

Table 2.8: Number of satisfiable and unsatisfiable instances solved faster per encoding at their corresponding time range.

lambdas in our case. As an example, Table 2.10 shows the percentage of satisfiable instances and the computation time to solve 20 instances of a problem with 10 nodes, 20 traffic demands, $\alpha = 0.65$ and $\beta = 1$, for different values of λ. As we focus on network sustainability, as traffic demands grow, we must be able to study and solve the RWA problem under regimes at which the number of resources is at a critical level.

With these goals in mind, we have designed our problem instance generator to obtain a wide range of problem instances. From instances with a high number of solutions, to instances with no solutions, passing trough an intermediate class of instances in which the level of resources is at a critical level. Such instances have very few solutions and are very hard to solve, especially with greedy algorithms. With our generator, these different classes of instances are easily generated by simply modifying the parameter λ as we keep fixed the other parameters.

SAT and PB Encodings Performance

Table 2.8 summarizes the performance of our four different basic PB encodings (PB_{2+3}, PB_{2+3b}, PB_{2b+3}, and PB_{2b+3b}), our compact PB encoding (PB_{cpt}) and the SAT encoding (SAT).[7] The results are presented separated

[7]The solver used for SAT solving, as well as solving CNF encoding of PB encodings is Precosat [Bie]. The CNF formulas are generated from the corresponding PB encodings using Minisat+ using sorters [ES06].

for instances with solution (left table) and for instances with no solution (right table). Each column shows the number of instances that the corresponding encoding has solved faster. The timeout to stop the SAT solver, with any encoding, has been 23 hours per instance. Even with such timeout, some instances have not been solved with some of the encodings.

For satisfiable instances, the encoding $\text{PB}_{2+3\text{b}}$ has the best performance for the easiest instances, but as the difficulty increases, the SAT and PB_{cpt} encodings perform better. Overall, we observe that the encoding $\text{PB}_{2+3\text{b}}$ does better although followed closely by the SAT encoding. The PB_{cpt} encoding only gives an advantage for some difficult satisfiable instances.

For unsatisfiable instances, that represent cases with not enough resources to satisfy all traffic demands, we again observe that $\text{PB}_{2+3\text{b}}$ performs well for easy instances, but when difficulty increases PB_{cpt} improves. We believe that this is due to the more compact form of representing assignment of wavelengths to routes in PB_{cpt}, such that there are no partial solutions with more than one wavelength assigned to the same path of a route. This fact may allow the SAT solver to reach more quickly the branches of the search tree that represent different assignments of wavelengths to the routes. However, given the higher size of the constraint that avoids using the same wavelength in different conflicting routes in PB_{cpt}, $O(\lambda|E||R|^2)$, with respect to the one of $\text{PB}_{2+3\text{b}}$, $O(\lambda|E||R|\log^2(|R|))$, the possible improvement thanks to a more compact search space may be reduced due to the increased time needed to check the constraints.

To compare the relative performance of our best encodings, $\text{PB}_{2+3\text{b}}$ and PB_{cpt}, with the SAT encoding, in a more quantitative way, we have also created scatter plots, where every point (x, y) in a plot represents one of the instances of the test-set with the x value representing its solving time with one encoding and the y value its solving time with a second encoding. Figure 2.9 shows a scatter plot between the SAT and the PB_{cpt} encodings (left plot) and a scatter plot between the SAT and the $\text{PB}_{2+3\text{b}}$ encodings (right plot). We observe that PB_{cpt} is either competitive or superior (especially for unsatisfiable instances) to the SAT encoding. $\text{PB}_{2+3\text{b}}$ seems to be clearly superior to the SAT encoding on unsatisfiable instances, but for satisfiable instances, the SAT encoding is almost always the best. However, the experiments at the end of this section show that as the number of demands increases, even for satisfiable instances, $\text{PB}_{2+3\text{b}}$ starts to outperform the SAT encoding.

We have also tested the performance of the best solving approach reported in [ZTTD03], named NEW algorithm in that reference, that can be seen as an improved version of the common approach used to solve the RWA problem in two phases with a greedy approach. First, a transformed graph is used to determine the shortest route between source and destination using the Dijkstra shortest path algorithm, where the weights of the links quantify the probability for a request to pass through a node, and later a simple

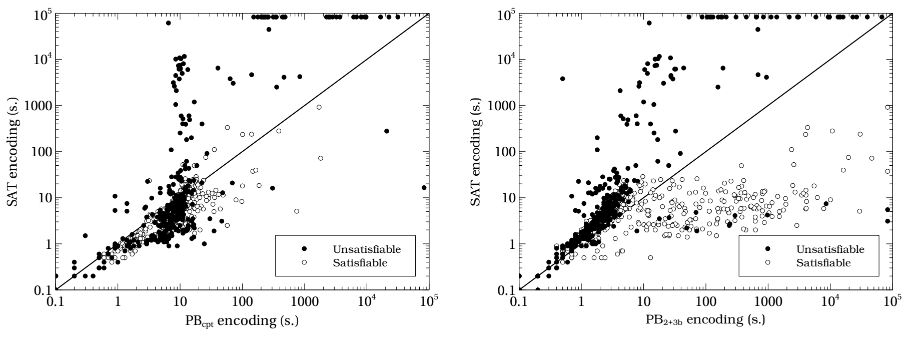


Figure 2.9: Scatter plot of times to solve satisfiable and unsatisfiable instances for different SAT based encodings.

greedy scheme is used to assign a proper wavelength to that path. Once a wavelength is assigned, the weights of the routing graph are updated, in order to attend future demands. Although it is an adaptive algorithm that tries to reduce the blocking probability, its greedy nature makes it very inefficient for RWA-SLE problem because, when considering the combined problem (routing + wavelength assignment), it is possible that it is unable to find a solution for a problem with a certain number of wavelengths, although the problem is perfectly solvable.

Our tests show that for a given network and demand set, our approach needs less wavelengths to satisfy them. As an example, for the same network with 10 nodes reported above in Table 2.10, the NEW algorithm only finds satisfiable instances for λ above 10, and even for $\lambda = 10$, the percentage of satisfiable found instances is a scarce 10%. As greedy algorithms perform better for a large λ, we conduce some additional tests for a larger λ and more traffic demands. Concretely, we take 20 nodes, $a = 0.4$, $b = 0.5$, and $\lambda = 20$. Table 2.9 shows the time to solve an instance for different values of traffic demands. Note that all the instances are satisfiable.

From Table 2.9 one can derive some facts. First, NEW algorithm is not able to find solutions when demands go beyond 25. Greedy algorithms do not do particularly good when resources are scarce. Second, due to the encoding size, the SAT encoding is not able to solve even the smaller case, 25 demands, because of memory exhaustion (1 GB of RAM). The best performing PB encoding is PB$_{2+3b}$, which is able to solve up to 100 traffic demands.

As a final remark we can say that we studied the use of current SAT solvers for the resolution of the RWA-SLE problem. We proved that a good formulation allocates network resources more efficiently than other approaches, as greedy algorithms, extensively studied in the specialized literature, at least, for more critically constrained problems. We also contributed new PB encoding variants, highly competitive with the existing SAT formu-

	Time(s.)			
# demands	PB_{cpt}	PB_{2+3b}	SAT	NEW
25	10	8	-	1
50	-	20	-	u
75	-	123	-	u
100	-	1,395	-	u
125	-	-	-	u

Table 2.9: Time in seconds to solve an instance for 20 nodes, $a = 0.4$, $b = 0.5$, and $\lambda = 20$. (-) indicates memory exhaustion, and (u) unsatisfiable reported.

λ	% sat	Mean time (s.)
2	40	0.59
3	60	0.79
4	70	192.39
5	90	1.58
6	90	2.18

Table 2.10: An example of easy-hard-easy hardness characterization on RWA problems.

lation for this problem, that make easier the task of extending the proposed formulation to RWA problems with partial lambda translation. And finally, we reported a comparative performance test among our encodings, SAT encodings and greedy algorithms under different network scenarios.

2.4.4. ASP-Based Encodings

In this section we present an Answer Set Programming (ASP) solving approach for solving the RWA problem for its SLE variant (RWA-SLE). Answer Set programming has become an attractive tool for representation and reasoning. Although some solvers were proposed in [SNS02, LPF+06], recent work in ASP solving has incorporated techniques such as conflict driven learning, backjumping, restart and watched literals lists in the state-of-the-art ASP solver Clasp [GKNS07]. This ASP solver has been the winner in many categories of benchmarks in the past ASP competitions [GLN+07, DVB+09, CIR+11].

We propose and study four different ASP encodings for RWA-SLE, that differ in the way of encoding some of the constraints of the problem. The main difference between these alternative encodings is basically the use of integrity constraints with respect to the use of regular clauses. We study how a measure of the tightness of the programs we propose relates to its relative performance and to the number of generated loop nogoods during search.

Our results indicate that incorporating measures of the tightness of ASP encodings could be a good predictive tool for selecting the best ASP encoding of a problem. We also compare this new ASP encoding approximation for RWA-SLE problem with the current best performing pseudo-Boolean based approach from Section 2.4.2. The experimental results show that our ASP approach can be better or competitive with the PB based approach.

ASP Encoding

We present an ASP encoding for RWA-SLE that follows an analogous encoding scheme than the best pseudo-Boolean encoding we have found for it, that is the PB_{2+3b} encoding we have explained in the Section 2.4.2. So, our ASP encoding is based on the following constraints:

1. The starting and ending nodes of the route must be active. Two constraints assures that the Boolean variable representing source and destination are evaluated to True.

2. On a route, there is only one active link connected with *src* and *dst*.

3. For any node k, different from *src* and *dst*, either it is active together with two adjacent links or it is not active and none of its links are active.

4. For any intermediate node j in a route, if a wavelength w is used in link $l_{i,j}$ for a given route, then it must also be used in adjacent link $l_{j,k}$ that is active in the same route.

5. A wavelength w is used by at most one of the route demands in a link.

However, taking profit of the different ways of modelling some of the constraints in the ASP framework we have used, Clingo, we will present four variants of the ASP encoding for trying to identify key aspects in the performance of ASP encodings.

Following the procedure described in [GKK$^+$11a], an ASP encoding is specified with a set of rules with non-ground predicates (predicates with variables as arguments) such that the data of a particular problem instance will be specified later as a set of ground instantiations of some of the predicates of the encoding.

We will describe some alternatives for encoding some of the constraints that will give place to four different alternative ASP encodings. Observe that from an ASP encoding with non ground rules and the set of ground predicates that define a problem instance we need to finally create an ASP instance with only ground predicates. This is done automatically by a grounder used by the ASP system Clingo, and once the grounded ASP instance is obtained, Clingo solves it with the ASP solver Clasp.

We use some more expressive rules beyond the classical rules of normal logic programs, because they are allowed by the ASP system Clingo. These are the additional kinds of rules:

- An *integrity constraint*, that is of the form

$$\bot \leftarrow p_0, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n$$

  and is actually a way to express directly a nogood on the logic program.

- A *choice rule* that is of the form

$$h_0, \ldots, h_k \leftarrow p_0, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n$$

  allows for the non-deterministic choice over atoms in $h_1, \ldots, h_k$.

- A *cardinality rule* of the form

$$h \leftarrow k_1\{p_0, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\}k_2$$

  infers h if the number of satisfied literals from

$$\{p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\}$$

  is in $[k_1, k_2]$.

- A *cardinality choice rule* of the form

$$k_1\{h_0, \ldots, h_k\}k_2 \leftarrow p_0, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n$$

  allows for the non-deterministic choice of a subset S of atoms from $h_1, \ldots, h_k$ such that $k_1 \geq |S| \geq k_2$.

For a more detailed explanation of the semantics of these extended kinds of rules, we refer the reader to [GKK$^+$08].

These are the different ASP non ground rules of our ASP encoding for RWA-SLE:

1. For each connection demand $r_{s,t}$ we define predicates that mark s as the source node of the connection and t as the termination node with the predicates $source(s, i)$ and $destination(t, i)$, being i the identifier of the demand. Although this information is already included in the predicate $connection(S, T, C)$, this helps to simplify the modelling of the next constraint. This constraint is modelled as the next cardinality choice rule and is analogous to constraint p1 in PB_{2+3b}:

$$2\ \{source(S, C), destination(T, C)\}\ 2 \leftarrow \quad connection(S, T, C).$$

2. We establish a connected lightpath for any connection demand C using the predicate $use\_edge(A, B, C)$. First, we use cardinality choice rules that force source and termination nodes to be connected with exactly one other node in the lightpath for C:

$$1\ \{use\_edge(S, A, C) \mid edge(S, A), node(A)\}\ 1 \leftarrow source(S, C).$$

$$1\ \{use\_edge(B, T, C) \mid edge(B, T), node(B)\}\ 1 \leftarrow destination(T, C).$$

   Secondly, any other node, that is not the starting or termination node of a connection demand C, if it is connected to one node in the lightpath for C then it must be connected to exactly one other node. We force this with the following cardinality choice rules:

$$\begin{array}{l} 1\{use\_edge(B, X, C) \mid edge(B, X), node(X)\}1 \quad \leftarrow \\ use\_edge(A, B, C), T \neq B, destination(T, C). \\ 1\{use\_edge(X, A, C) \mid edge(X, A), node(X)\}1 \quad \leftarrow \\ use\_edge(A, B, C), S \neq A, source(S, C1). \end{array}$$

   These constraints are related to constraints p2,p3 and p4 in PB_{2+3b} Also, as the lightpath for a connection C from S to T will be encoded as a directed path starting at S and ending at T, and edge $\{A, B\}$ can only be used in one direction, so we use the following integrity constraint:

$$\perp \leftarrow use\_edge(A, B, C), use\_edge(B, A, C).$$

3. We add redundant rules to force that any node B used in a lightpath cannot be the destination of two incoming nodes $A1$ and $A1$ or be the origin of two outgoing nodes $A1$ and $A2$. We say that they are redundant because the rules we have introduced in the previous step, and considering that answer sets are always well-founded models, do not allow to have more than one incoming or outgoing connection with a node. Adding redundant clauses, specially integrity constraints that are directly equivalent to nogoods, can improve the performance of an ASP solver like Clasp, given that the inference mechanism of unit propagation it uses is based on the discovery of forced unit assignments obtained from nogoods and the current partial solution.

$$\begin{array}{l} \perp \leftarrow use\_edge(A2, B, C), use\_edge(A1, B, C), A1 \neq A2. \\ \perp \leftarrow use\_edge(B, A2, C), use\_edge(B, A1, C), A1 \neq A2. \end{array}$$

   Also, as redundant rules, we force with integrity constraints that the source (S) and termination (T) nodes of a lightpath for C are not connected to any other nodes than the ones specified with the previous rules:

$$\begin{array}{l} \perp \leftarrow source(S, C), use\_edge(A, S, C). \\ \perp \leftarrow destination(T, C), use\_edge(T, B, C). \end{array}$$

4. We add a cardinality choice rule to force that for each used directed edge $A \to B$ for connection C ($use\_edge(A,B,C)$ is true) exactly one predicate $use\_link(A,B,C,L)$ must be true, among all possible lambdas we can use for the link of the edge. So, the predicate $use\_link(A,B,C,L)$ indicate the lambda used in the link of the edge for that connection.

$$\begin{array}{l} 1\{use\_link(A,B,C,L) | L \in [1..\Lambda]\}1 \leftarrow \\ \quad use\_edge(A,B,C1). \end{array}$$

5. We need to force that the same lambda is used in all the links of a lightpath. For forcing this constraint we propose two alternatives ways for encoding it.

   a) A rule for forcing that when directed edges $A \to B$ and $B \to X$ are used and L is used for the link of $A \to B$ ($use\_link(A,B,C,L)$ is true), then the same lambda L must be used in the link for $B \to X$:

   $$\begin{array}{l} use\_link(B,X,C,L) \leftarrow use\_link(A,B,C,L), \\ \quad use\_edge(A,B,C), use\_edge(B,X,C). \end{array}$$

   b) We can instead force the same property with the following integrity constraint:

   $$\begin{array}{l} \bot \leftarrow not\ use\_link(B,X,C,L), \\ \qquad use\_link(A,B,C,L), use\_edge(A,B,C), \\ \qquad use\_edge(B,X,C). \end{array}$$

   As a first analysis of the difference these two alternatives for encoding the constraint can make, observe that 5a is a rule where a same predicate name (use_link) appears both at the head and at the body. So, if the network $G = (N,E)$ of the problem instance is connected, as in our test sets of the experimental section, these rules will create loops in the dependency graph G_Π of the logic program. By contrast, with the integrity constraint 5b such problem is not present. Observe that the rules of the item 2 for forcing connected lightpaths also create loops in G_Π.

6. Because we allow to use an edge $\{A,B\}$ in only one direction, $A \to B$ or $B \to A$, we could also force the corresponding predicates use_link (A,B,C,L) and $use\_link(B,A,C,L)$ to be not both true. However, to simplify the encoding of the next constraint, we are going to force the contrary condition, both predicates will be true or both will be false with the following rule, even if the edge is used only in one direction:

$$use\_link(B,A,C,L) \leftarrow use\_link(A,B,C,L).$$

7. We need to force that the same lambda is not used by two or more lightpaths that share at least one link. For encoding this constraint we have also considered two alternatives:

   a) An integrity constraint for not allowing to have a same link $A-B$ used with a same lambda L by any two different connection demands $C1$ and $C2$

   $$\begin{aligned}\bot \leftarrow\ & use\_link(A,B,C1,L),\\ & use\_link(A,B,C2,L), C1! = C2.\end{aligned}$$

   b) In a very similar way, for any link $A-B$ and lambda L we can use instead a cardinality integrity constraint that does not allow to have more than one predicate $use\_link(A,B,C,L)$ true in the set $\{use\_link(A,B,C,L)|C \in [0,|R|]\}$

   $$\begin{aligned}\bot \leftarrow\ & |R|\ \{use\_link(A,B,C,L)|C \in [0,|R|]\}\ 2,\\ & edge(A,B), node(A), node(B).\end{aligned}$$

   It is not clear what of the two options should be better, as both are integrity constraints, but the second one is a more complex one that must be transformed by the grounder of Clingo to a set of more basic rules [8].

8. Finally, we add some other redundant integrity constraints:

   $$\begin{aligned}\bot \leftarrow\ & not\ use\_link(B,T,C,L), source(S,C),\\ & use\_link(S,A,C,L), destination(T,C), use_edge(B,T,C).\end{aligned}$$

   $$\begin{aligned}\bot \leftarrow\ & not\ use\_link(S,A,C,L), destination(T,C),\\ & use\_link(B,T,C,L), source(S,C), use\_edge(S,A,C).\end{aligned}$$

Given the two possible ways of encoding constraint 5 (5a and 5b) and the two possible ways of encoding constraint 7 (7a and 7b) that we have presented, we can define four different ASP encodings for RWA-SLE, depending on the option selected for constraints 5 and 7. So, we identify four ASP encodings $ASP_{5a7b}, ASP_{5b7b}, ASP_{5b7a}$ and ASP_{5a7a} that we use in the experimental section to study and compare their performance.

2.4.5. Experimental Analysis for ASP Based Encodings

In this section we compare the performance of the four different ASP encodings for solving RWA-SLE. We focus on randomly generated problems, as we have seen that the current state of RWA networks deployment makes

[8] We do not have information about the particular transformations that Clingo applies to these classes of more complex rules

real-world scenarios very easy to solve. Instead of real scenarios, we use synthetic problems like in Section 2.4.2, where we can tune some of the properties of the network and traffic demands. In fact, we use the same problem generator that we described in 2.4.2 but we will select a different range of scenarios.

We will compare the performance of our ASP encodings with the one obtained with the best PB encoding, that we found in Section 2.4.3, the PB_2+3b encoding that we have presented in Section 2.4.2.

The ASP encoded instances of RWA-SLE are solved with the ASP tool Clingo, that incorporates a grounder that given the non-ground rules based ASP encoding and a particular instance generates the resulting set of ground rules for the instance. Then, Clingo uses the Clasp ASP solver to solve the instance. In all the instances, we have used the Clasp decision heuristic VSIDS, that is similar to the one used by Minisat+.

Our experiments have been run on machines with the following specs: Rocks Cluster 5.2 Linux 2.6.18 Operating System, AMD With two Opteron 248 Processor clocked at 1.59 GHz, 1.0GB Memory, and GCC 4.1.2 Compiler.

Next we describe the scenarios where we believe that our ASP encodings can perform as better as PB approaches.

Selected Scenarios

Our aim is to generate problem instances by using different parameters in the RWA generator, going from very relaxed problem instances, where the number of demands with respect to the resources (paths and lambdas) is low so instances tend to have many solutions, to overconstrained problem instances, where the number of demands with respect to the resources is so high that instances tend to have no solutions.

We have generated different test sets of instances, each one with 50 instances. First, we have generated four sets with $\alpha = 0.4$ and $\beta = 0.25$, that give place to very sparse networks, with $\lambda = 3$ and with number of nodes equal to 16. The four sets differ then only on the number of demands in each one: $\{10, 15, 20, 25\}$. So, observe that by fixing the network size and average density of links and the number of lambdas but increasing the number of demands, we will be able to generate scenarios with different *constrainedness*. A similar set of four test sets is also generated but increasing now the number of nodes to 20.

Then, we generate new sets of instances as before, but now increasing β to 0.5 and changing the range of demands to set $\{15, 20, 25, 30\}$. This way, for the same number of demands and nodes, as instances will tend to have more paths, they will tend to have more solutions.

Finally, we have also generated analogous test sets of instances by now increasing λ to 5 but only with a number of nodes equal to 16.

ASP Encoding Performance

	$nodes = 16, \lambda = 3$								
demands		10		15		20		25	
$SAT(\%)$		18		2		0		0	
	SCC_{G_P}	time	nogoods	time	nogoods	time	nogoods	time	nogoods
ASP_{5a7b}	0.00	0.03	63-444	0.05	79-628	0.11	104-885	0.19	124-1148
ASP_{5b7b}	0.12	0.01	41-401	0.03	48-544	0.03	53-609	0.06	84-863
ASP_{5b7a}	0.12	0.02	43-386	0.01	10-255	0.01	7-170	0.01	7-42
ASP_{5a7a}	0.00	0.04	62-425	0.05	66-517	0.06	64-512	0.05	35-456
PB		0.03		0.06		0.11		0.20	

	$nodes = 20, \lambda = 3$								
demands		10		15		20		25	
$SAT(\%)$		42		12		2		0	
	SCC_{G_P}	time	nogoods	time	nogoods	time	nogoods	time	nogoods
ASP_{5a7b}	0.00	0.09	129-701	0.16	142-997	0.31	206-1304	0.41	197-1736
ASP_{5b7b}	0.12	0.06	129-687	0.14	145-934	0.17	125-1127	0.20	147-1165
ASP_{5b7a}	0.12	0.06	131-704	0.09	111-667	0.01	8-162	0.01	7-70
ASP_{5a7a}	0.00	0.09	120-679	0.18	148-898	0.20	121-1030	0.125	81-814
PB		0.04		0.09		0.16		0.26	

Table 2.11: Experimental results for instances with $\alpha = 0.4$, $\beta = 0.25$, $\lambda = 3$, $nodes = \{16, 20\}$ and $demands = \{10, 15, 20, 25\}$. Gray cells indicate the best results among the ASP encodings

As we have said before, our aim is to study and understand the performance of the different ASP encodings we have presented. Table 2.11 shows the results for test sets with $\lambda = 3$, $\alpha = 0.4$ and $\beta = 0.25$. The upper part shows the results for the four test sets with instances with 16 nodes, and the lower part the results for the four test sets with instances with 20 nodes. The column labelled SCC_{G_P} is the median value of the tightness we have measured for the instances of each test set. Although this value is not exactly the same for each one of the four test sets for a same ASP encoding, when rounded up to two decimal digits, as we have done here, it shows the same value for the four test sets, so we only show one column for this value. The column labelled time indicates the median time to solve an instance from a test set. The column labelled nogoods indicates a pair of numbers $n1 - n2$, where $n1$ is the median number of learned nogoods and $n2$ is the median number of generated loop nogoods. The row labelled SAT% indicates the percentage of instances that have solution in each test set.

We observe that the best ASP encoding is ASP_{5b7a} in three of the four test sets with 16 nodes and in the four test sets with 20 nodes, and that at the same time it also shows almost always the smaller number of learned nogoods and generated loop nogoods. Also, ASP_{5b7b} is the best one or the second best one in three of the four test sets. So, observe that the key difference seems to be the choice of option 5b for constraint 5 of the ASP encoding. That is, an integrity constraint instead of the regular rule of option 5a. This seems reasonable if we consider that the rule of option 5a, having the same predicate name `use_link` in both its body and its head, can easily create cycles in the dependence graph of the instances, as we have discussed before, and so increases the possibilities of having unfounded sets in partial solutions, thus giving place to loop nogoods. Instead, option 5b is an integrity constraint, so it cannot participate in any cyclic dependence between atoms. This is further supported by the fact that our tightness measure is higher for these two ASP encodings. It is also worth noticing that the number of generated loop nogoods is always the quantity that dominates the total number of nogoods, thus supporting our hypothesis that the generation of loop nogoods, and so the tightness of the instances, is an important factor regarding the performance of ASP encodings.

The reason of why option 7a seems to be a better option than 7b could be that although 7b is a cardinality integrity constraint similar to a regular integrity constraint, its concrete translation to a simpler set of integrity constraints or rules by the grounder used by Clingo may not be generating a set of integrity constraints as short as the ones of option 7a. Shorter nogoods are preferable to longer ones, because in the process of unit propagation performed by the ASP solver Clasp used by Clingo more unit literals will tend to be produced before if we have shorter nogoods. See [GKNS07] for a discussion of the unit propagation process performed with nogoods. Remember that a grounded integrity constraint is equivalent to a nogood.

With respect to the performance of the PB encoding, observe that the best ASP encoding is always better than or similar to the PB encoding. As the number of instances with solution decreases when we increase the number of demands on the test set, the advantage of ASP with respect to PB increases.

		$nodes = 16, \lambda = 3$							
demands		15		20		25		30	
$SAT(\%)$		44		20		12		2	
	SCC_{G_P}	time	nogoods	time	nogoods	time	nogoods	time	nogoods
ASP_{5a7b}	0.00	0.30	450-1186	0.44	332-1464	0.51	310-1846	0.53	280-2069
ASP_{5b7b}	0.12	0.17	352-1173	0.18	250-1020	0.25	186-826	0.36	241-1366
ASP_{5b7a}	0.12	0.18	372-1017	0.12	149-547	0.02	9-263	0.02	8-102
ASP_{5a7a}	0.00	0.36	475-1172	0.34	253-1189	0.17	86-895	0.09	43-650
PB		0.11		0.19		0.29		0.37	

		$nodes = 20, \lambda = 3$							
demands		15		20		25		30	
$SAT(\%)$		72		64		28		2	
	SCC_{G_P}	time	nogoods	time	nogoods	time	nogoods	time	nogoods
ASP_{5a7b}	0.00	2.04	1816-2085	7.46	5916-3597	1.60	1110-3066	1.60	671-3605
ASP_{5b7b}	0.12	1.34	1467-2133	4.46	5427-4018	0.87	955-2523	0.93	476-2672
ASP_{5b7a}	0.12	1.14	1824-2014	6.74	7694-4094	0.31	547-881	0.18	22-627
ASP_{5a7a}	0.00	3.41	2580-2249	8.17	6096-3587	1.30	765-2407	1.05	446-1921
PB		0.16		0.36		0.48		0.78	

Table 2.12: Experimental results for instances with $\alpha = 0.4$, $\beta = 0.5$, $\lambda = 3$, $nodes = \{16, 20\}$ and $demands = \{15, 20, 25, 30\}$.

Table 2.12 shows analogous results but now for test sets with $\lambda = 3$, $\alpha = 0.4$ and $\beta = 0.5$. As before, we observe that the best ASP encoding is ASP_{5b7a} and that at the same time it also shows almost always the smaller number of learned nogoods and generated loop nogoods and together with ASP_{5b7b} has the higher tightness. Observe also that the second best ASP encoding is ASP_{5b7b} in all test sets except on the two last ones, where almost all the instances are unsatisfiable, and that it has the same tightness value

		$nodes = 16, \lambda = 5$							
demands		10		15		20		25	
$SAT(\%)$		70		26		12		6	
	SCC_{G_P}	time	nogoods	time	nogoods	time	nogoods	time	nogoods
ASP_{5a7b}	0.00	0.09	217-753	0.17	319-1063	0.23	304-1391	0.40	304-1707
ASP_{5b7b}	0.15	0.05	170-725	0.11	275-1000	0.15	273-1239	0.24	251-1595
ASP_{5b7a}	0.15	0.04	139-712	0.07	245-809	0.05	153-362	0.06	156-239
ASP_{5a7a}	0.00	0.09	200-762	0.17	329-1055	0.24	317-1322	0.37	321-1371
PB		0.05		0.11		0.20		0.31	

Table 2.13: Experimental results for instances with $\alpha = 0.4$, $\beta = 0.25$, $\lambda = 5$, $nodes = \{16\}$ and $demands = \{10, 15, 20, 25\}$.

		$nodes = 16, \lambda = 5$							
demands		15		20		25		30	
$SAT(\%)$		82		78		58		28	
	SCC_{G_P}	time	nogoods	time	nogoods	time	nogoods	time	nogoods
ASP_{5a7b}	0.00	0.75	743-1919	2.43	1818-2768	8.28	310-1846	17.66	3088-3873
ASP_{5b7b}	0.15	0.42	669-1874	1.43	1986-2737	5.19	186-826	21.53	5911-3175
ASP_{5b7a}	0.15	0.40	654-1807	1.16	1500-2561	4.30	9-263	1.84	325-1301
ASP_{5a7a}	0.00	0.90	763-1918	2.92	2155-2743	12.22	86-895	46.08	11745-3521
PB		0.17		0.32		0.54		0.88	

Table 2.14: Experimental results for instances with $\alpha = 0.4$, $\beta = 0.5$, $\lambda = 5$, $nodes = \{16\}$ and $demands = \{15, 20, 25, 30\}$.

that ASP_{5b7a}. So, again it seems that the number of learned nogoods and generated loop nogoods are good indicators of the hardness of the search for a solution, and specially the number of loop nogoods that is always higher than the number of regular nogoods. The smaller number of loop nogoods is almost always obtained with the two ASP encodings with higher tightness.

The results we obtain here for PB show a similar behaviour than before, although that given that in these test sets we have a higher fraction of instances with solution, the advantage of ASP is only relevant in three of the four test sets for $nodes = 16$ and in two of the four test sets for $nodes = 20$.

Finally, we present the results for the test sets with $\lambda = 5$. Table 2.13 shows the results for test sets with $\lambda = 5$, $\alpha = 0.4$ and $\beta = 0.25$. As now we have more resources per link, for the same number of demands with respect to the results of Table 2.11 we obtain more instances with solution. As before, the best ASP encoding is ASP_{5b7a} that obtains the best results in median time and median number of loop nogoods. This time, the second best ASP encoding is always ASP_{5b7b}. So, for these test sets the two best ASP encodings are the ones with the higher tightness.

Table 2.14 shows the results for $\lambda = 5$, $\alpha = 0.4$ and $\beta = 0.5$. Again, ASP_{5b7a} is the best ASP encoding but ASP_{5b7b} is the second one in three of the four test sets. With respect to the performance of the PB encoding, we also obtain similar results: ASP encodings are competitive or better than the PB encoding.

Observe that even if one could think, given our results, that using integrity constraints instead of regular rules in ASP encodings should be always the best option, there is a drawback in the excessive use of integrity constraints. The problem is that using regular rules sometimes we can get more compact encodings than using sets of integrity constraints. So, one should try to get a balance between encoding size and solving time performance.

As a final remark we can say that we studied the use of ASP solvers for the resolution of the RWA-SLE problem. We contributed with new ASP encodings for the problem, and we studied how the tightness and the number of generated loop nogoods relate with the complexity of solving the instances. In addition, a comparison with the best previous pseudo-Boolean encoding has shown that the ASP approach is quite competitive with the well established approach based on PB solvers that use SAT encodings and state-of-the-art CDCL SAT solvers. Our tightness measure for ASP instances indicates that there is probably still room for improvement in the ASP encodings for this problem, if ASP encodings with better tightness, but still of compact size, can be found.

3 Recursive Defeasible Logic Programming

In this chapter we present a particular framework of Possibilistic Defeasible Logic Programming (RP-DeLP for short). This framework is based on a general notion of collective (non-binary) conflict among arguments allowing to ensure direct and indirect consistency properties with respect to the strict knowledge. An output of an RP-DeLP program is a pair of sets of warranted and blocked conclusions (literals), all of them recursively based on warranted conclusions but, while warranted conclusions do not generate any conflict, blocked conclusions do. An RP-DeLP program may have multiple outputs in case of circular definitions of conflicts among arguments.

Contents

3.1. Introduction and Motivation

Defeasible argumentation is a natural way of identifying relevant assumptions and conclusions for a given problem which often involves identifying conflicting information, resulting in the need to look for pros and cons for a particular conclusion [PV02]. This process may involve chains of reasoning, where conclusions are used in the assumptions for deriving further conclusions and the task of finding pros and cons may be decomposed recursively. Logic-based formalizations of argumentation that take pros and cons for some conclusion into account assume a set of formulas and then lay out arguments and counterarguments that can be obtained from these assumed formulas [BH08].

Defeasible Logic Programming (DeLP) [GS04b] is a formalism that combines techniques of both logic programming and defeasible argumentation. As in logic programming, knowledge is represented in DeLP using facts and rules; however, DeLP also provides the possibility of representing defeasible knowledge under the form of weak (defeasible) rules, expressing reasons to believe in a given conclusion. In DeLP, a conclusion succeeds if it is warranted, i.e., if there exists an argument (a consistent sets of defeasible rules) that, together with the non-defeasible rules and facts, entails the conclusion, and moreover, this argument is found to be undefeated by a warrant procedure which builds a dialectical tree containing all arguments that challenge this argument, and all counterarguments that challenge those arguments, and so on, recursively. Actually, dialectical trees systematically explore the universe of arguments in order to present an exhaustive synthesis of the relevant chains of pros and cons for a given conclusion. In fact, the interpreter for DeLP [GDS09] (http://lidia.cs.uns.edu.ar/DeLP) takes a knowledge base (program) P and a conclusion (query) Q as input, and it then returns one of the following four possible answers: YES, if Q is warranted from P; NO, if the complement of Q is warranted from P; UNDECIDED, if neither Q nor its complement are warranted from P; or UNKNOWN, if Q is not in the language of the program P.

Possibilistic Defeasible Logic Programming (P-DeLP) [ACGS08] is an extension of DeLP in which defeasible rules are attached with weights (belonging to the real unit interval $[0, 1]$) expressing their relative belief or preference strength. As many other argumentation frameworks [CML00, PV02], P-DeLP can be used as a vehicle for facilitating rationally justifiable decision making when handling incomplete and potentially inconsistent information. Actually, given a P-DeLP program, justifiable decisions correspond to warranted conclusions (to some necessity degree), that is, those which remain undefeated after an exhaustive dialectical analysis of all possible arguments for and against.

In [CA07] Caminada and Amgoud proposed three *rationality postulates* which every rule-based argumentation system should satisfy. One of such postulates (called *Indirect Consistency*) requires that the set of warranted conclusions must be consistent (wrt the underlying logic) with the set of strict facts and rules. In [CA07] a number of rule-based argumentation systems were identified in which such postulate does not hold (including DeLP [GS04b] and Prakken & Sartor's [PS97], among others). As a way to solve this problem, the use of *transposed rules* is proposed in [CA07] to extend the representation of strict rules. Recently, in [Amg12] Amgoud proposes a new rationality postulate (called *Closure under Subarguments*) which rule-based argumentation systems should satisfy. This postulate claims that the acceptance of an argument should imply also the acceptance of all its subarguments which reflect the different premises on which the argument is based.

Since the dialectical analysis-based semantics of (P-)DeLP for warranted conclusions does not satisfy the Indirect Consistency postulate, in [ABG10] Alsinet *et. al.* provide (P-)DeLP with a new semantics satisfying the above mentioned postulates.

To this end, in [ABG10] recursive semantics for defeasible argumentation is considered as defined by Pollock in [Pol09], where recursive definitions of conflict between arguments were characterized by means of *inference-graphs*, representing (binary) support and attack (pros and cons) relations among the conclusions of arguments. Recursive semantics presented in [ABG10] are based on the fact that if an argument is rejected, then all arguments built on top of it should also be rejected. On the other hand, as stated in [Pol09], recursive definitions of conflict among arguments can lead to different outputs (extensions) for warranted conclusions.

In this chapter we introduce the recursive semantics with defeasibility levels to the particular framework of P-DeLP, this formalism is referred as *Recursive* P-DeLP (RP-DeLP for short).

Following the approach of Pollock [Pol09], in RP-DeLP inferences from some propositions to others and definitions of conflict are characterized by means of what the authors call *Warrant Dependency Graphs*, representing support and (collective) conflict relations between argument conclusions.

As stated in the introduction chapter, the one contribution of the thesis is to design and implement the argumentation framework behind RP-DeLP.

3.2. Weighted Recursive Semantics of RP-DeLP

The *language* of RP-DeLP, denoted $\mathcal{L}$, is inherited from the language of logic programming, including the notions of atom, literal, rule and fact. Formulas are built over a finite set of propositional variables $\{p, q, \ldots\}$ which is extended with a new (negated) atom "$\sim p$" for each original atom p. Atoms of the form p or $\sim p$ will be referred as literals.[1] *Formulas* of $\mathcal{L}$ consist of rules of the form $Q \leftarrow P_1 \wedge \ldots \wedge P_k$, where $Q, P_1, \ldots, P_k$ are literals. A fact will be a rule with no premises. The name *clause* is used to denote a rule or a fact. The R-DeLP framework is based on the propositional logic $(\mathcal{L}, \vdash)$ where the inference operator $\vdash$ is defined by instances of the modus ponens rule of the form: $\{Q \leftarrow P_1 \wedge \ldots \wedge P_k, P_1, \ldots, P_k\} \vdash Q$. A set of clauses Γ will be deemed as *contradictory*, denoted $\Gamma \vdash \bot$, if , for some atom q, $\Gamma \vdash q$ and $\Gamma \vdash \ \sim q$.

An RP-DeLP *program* $\mathcal{P}$ is a tuple $\mathcal{P} = (\Pi, \Delta, \preceq)$ over the logic $(\mathcal{L}, \vdash)$, where $\Pi, \Delta \subseteq \mathcal{L}$, and $\Pi \not\vdash \bot$. Π is a finite set of clauses representing strict knowledge (information that is taken as granted they hold true), Δ is another finite set of clauses representing the defeasible knowledge (formulas

[1]For a given literal Q, is written $\sim Q$ as an abbreviation to denote "$\sim q$" if $Q = q$ and "q" if $Q = \sim q$.

for which there are reasons to believe they are true). Finally, $\preceq$ is a total pre-order on $\Pi \cup \Delta$ representing levels of defeasibility: $\varphi \prec \psi$ means that φ is more defeasible than ψ. Actually, since formulas in Π are not defeasible, $\preceq$ is such that all formulas in Π are at the top of the ordering. For the sake of a simpler notation we will often refer in the paper to numerical levels for defeasible clauses and arguments rather than to the pre-ordering $\preceq$, so we will assume a mapping $N : \Pi \cup \Delta \to [0, 1]$ such that $N(\varphi) = 1$ for all $\varphi \in \Pi$ and $N(\varphi) < N(\psi)$ iff $\varphi \prec \psi$. [2] The notion of *argument* is the usual one. Given an RP-DeLP program $\mathcal{P} = (\Pi, \Delta, \preceq)$, an argument for a literal (conclusion) Q of $\mathcal{L}$ is a pair $\mathcal{A} = \langle A, Q\rangle$, with $A \subseteq \Delta$ such that $\Pi \cup A \not\vdash \perp$, and A is minimal (w.r.t. set inclusion) such that $\Pi \cup A \vdash Q$. If $A = \emptyset$, then $\mathcal{A}$ is called an s-argument (s for strict), otherwise it will be a d-argument (d for defeasible). The *strength of an argument* $\langle A, Q\rangle$, written $s(\langle A, Q\rangle)$, is defined as follows:

(i) $s(\langle A, Q\rangle) = 1$ if $A = \emptyset$; and

(ii) $s(\langle A, Q\rangle) = \min\{N(\psi) \mid \psi \in A\}$, otherwise.

The notion of *subargument* is referred to d-arguments and expresses an incremental proof relationship between arguments which is defined as follows. Let $\langle B, Q\rangle$ and $\langle A, P\rangle$ be two d-arguments such that the minimal sets (w.r.t. set inclusion) $\Pi_Q \subseteq \Pi$ and $\Pi_P \subseteq \Pi$ such that $\Pi_Q \cup B \vdash Q$ and $\Pi_P \cup A \vdash P$ verify that $\Pi_Q \subseteq \Pi_P$. Then, $\langle B, Q\rangle$ is a *subargument* of $\langle A, P\rangle$, written $\langle B, Q\rangle \sqsubset \langle A, P\rangle$, when either $B \subset A$ (strict inclusion for defeasible knowledge), or $B = A$ and $\Pi_Q \subset \Pi_P$ (strict inclusion for strict knowledge). A literal Q of $\mathcal{L}$ is called *justifiable conclusion* w.r.t. $\mathcal{P}$ if there exists an argument for Q, i.e., there exists $A \subseteq \Delta$ such that $\langle A, Q\rangle$ is an argument.

The warrant recursive semantics for RP-DeLP is based on the following notion of collective conflict in a set of arguments which captures the idea of an inconsistency arising from a consistent set of justifiable conclusions W together with the strict part of a program and the set of conclusions of those arguments. Let $\mathcal{P} = (\Pi, \Delta, \preceq)$ be an RP-DeLP program and let $W \subseteq \mathcal{L}$ be a set of conclusions. A set of arguments $\{\langle A_1, Q_1\rangle, \ldots, \langle A_k, Q_k\rangle\}$ *minimally conflicts* with respect to W iff the two following conditions hold:

(i) the set of argument conclusions $\{Q_1, \ldots, Q_k\}$ is contradictory with respect to W, i.e. it holds that $\Pi \cup W \cup \{Q_1, \ldots, Q_k\} \vdash \perp$; and

(ii) the set $\{\langle A_1, Q_1\rangle, \ldots, \langle A_k, Q_k\rangle\}$ is minimal with respect to set inclusion satisfying (i), i.e. if $S \subsetneq \{Q_1, \ldots, Q_k\}$, then $\Pi \cup W \cup S \not\vdash \perp$.

This general notion of conflict is used to define an output for an RP-DeLP program $\mathcal{P} = (\Pi, \Delta, \preceq)$ as a pair $(\mathit{Warr}, \mathit{Block})$ of subsets of $\mathcal{L}$ of

[2]Actually, a same pre-order $\preceq$ can be represented by many mappings, but we can take any of them since only the relative ordering is what actually matters.

warranted and blocked conclusions respectively all of them based on warranted information but while warranted conclusions do not generate any conflict, blocked conclusions do.

Since considering several levels of strength among arguments are considered, the intended construction of the sets of conclusions *Warr* and *Block* is done level-wise, starting from the highest level and iteratively going down from one level to next level below. If $1 > \alpha_1 > \ldots > \alpha_p \geq 0$ are the strengths of d-arguments that can be built within $\mathcal{P}$, one can define: $\mathit{Warr} = \mathit{Warr}(1) \cup \{\cup_{i=1,p} \mathit{Warr}(\alpha_i)\}$ and $\mathit{Block} = \cup_{i=1,p} \mathit{Block}(\alpha_i)$, where $\mathit{Warr}(1) = \{Q \mid \Pi \vdash Q\}$, and $\mathit{Warr}(\alpha_i)$ and $\mathit{Block}(\alpha_i)$ are respectively the sets of the warranted and blocked justifiable conclusions of strength α_i and are required to satisfy the following recursive constraints: [3]

1. $Q \in \mathit{Warr}(\alpha_i) \cup \mathit{Block}(\alpha_i)$ iff there exists an argument $\langle A, Q\rangle$ of strength α_i satisfying the following three conditions:

   (V1) for each subargument $\langle E, P\rangle \sqsubset \langle A, Q\rangle$ of strength β, $P \in \mathit{Warr}(\beta)$;

   (V2) $Q \notin \mathit{Warr}(> \alpha_i) \cup \mathit{Block}(> \alpha_i)$;

   (V3) $\sim Q \notin \mathit{Block}(> \alpha_i)$ and $\Pi \cup \mathit{Warr}(> \alpha_i) \cup \{P \mid \langle E, P\rangle \sqsubset \langle A, Q\rangle\} \cup \{Q\} \not\vdash \perp$.

   In this case is said that $\langle A, Q\rangle$ is *valid* with respect to the sets $\mathit{Warr}(\geq \alpha_i)$ and $\mathit{Block}(> \alpha_i)$.

2. For every valid argument $\langle A, Q\rangle$ of strength α_i

   - $Q \in \mathit{Block}(\alpha_i)$ whenever there exists a set $\mathbb{G}$ of valid arguments of strength α_i such that
     (i) $\langle A, Q\rangle \not\sqsubset \mathbb{G}$, and
     (ii) $\mathbb{G} \cup \{\langle A, Q\rangle\}$ minimally conflicts with respect to the set $W = \mathit{Warr}(> \alpha_i) \cup \{P \mid \langle E, P\rangle \sqsubset \mathbb{G} \cup \{\langle A, Q\rangle\}\}$.
   - otherwise, $Q \in \mathit{Warr}(\alpha_i)$.

Intuitively, an argument $\langle A, Q\rangle$ is valid whenever (V1) it is based on warranted conclusions; (V2) there does not exist a valid argument for Q with greater strength; and (V3) Q is consistent with already warranted and blocked conclusions. Then, a valid argument $\langle A, Q\rangle$ becomes blocked as soon as it leads to some conflict among valid arguments of same strength and the set of already warranted conclusions, otherwise it is warranted.

Next we outline some relevant properties regarding warranted and blocked conclusions when considering stratified strengths of arguments:

[3] In what follows we will also write $\mathit{Warr}(\geq \alpha_i)$ and $\mathit{Warr}(> \alpha_i)$ to denote $\cup_{\beta \geq \alpha_i} \mathit{Warr}(\beta)$ and $\cup_{\beta > \alpha_i} \mathit{Warr}(\beta)$, respectively, and analogously for $\mathit{Block}(> \alpha_i)$, assuming $\mathit{Block}(> \alpha_1) = \emptyset$.

1. If $Q \in \mathit{Warr}(\alpha) \cup \mathit{Block}(\alpha)$, then there exists an argument $\langle A, Q\rangle$ of strength α such that for all subargument $\langle E, P\rangle \sqsubset \langle A, \varphi\rangle$ of strength β, $\psi \in \mathit{Warr}(\beta)$.

2. If $Q \in \mathit{Warr}(\alpha) \cup \mathit{Block}(\alpha)$, then for any argument $\langle A, Q\rangle$ of strength β, with $\beta > \alpha$, there exists a subargument $\langle E, P\rangle \sqsubset \langle A, Q\rangle$ of strength γ and $P \notin \mathit{Warr}(\gamma)$.

3. If $Q \in \mathit{Warr}$, then $Q, \sim Q \notin \mathit{Block}$.

4. If $Q \notin \mathit{Warr} \cup \mathit{Block}$, then either $\sim Q \in \mathit{Block}$, or for all argument $\langle A, Q\rangle$ there exists a subargument $\langle E, P\rangle \sqsubset \langle A, Q\rangle$ such that $P \notin \mathit{Warr}$ or $\Pi \cup \mathit{Warr}(> \alpha_i) \cup \{P \mid \langle E, P\rangle \sqsubset \langle A, Q\rangle\} \cup \{Q\} \vdash \bot$.

Example 1. *Consider the RP-DeLP program $\mathcal{P}_1 = (\Pi, \Delta, \preceq)$, with*

$$\Pi = \{a \wedge b \wedge c \rightarrow y, \sim y\},\ \Delta = \{a, b, a \wedge b \rightarrow c\}.$$

Assume one defeasible level α_1 for Δ, so $1 > \alpha_1 > 0$. Obviously, $\mathit{Warr}(1) = \{\sim y\}$ and, at level α_1, arguments $\mathcal{A}_1 = \langle\{a\}, a\rangle$ and $\mathcal{A}_2 = \langle\{b\}, b\rangle$ are valid, and thus, conclusions a and b may be warranted or blocked but not rejected. At this point we can assure that $\mathit{Warr}(\alpha_1) = \{a, b\}$ as $\mathit{Warr}(> \alpha_1) \cup \{a, b\} \not\vdash \bot$. Next, argument $\mathcal{A}_3 = \langle\{a, b, a \wedge b \rightarrow c\}, c\rangle$ now is satisfying condition V1, but it minimally conflicts with the set $\mathit{Warr}(> \alpha_1)$, so $\mathit{Block}(\alpha_1) = \{c\}$. Argument $\mathcal{A}_4 = \langle\{a, b, a \wedge b \rightarrow c, a \wedge b \wedge c \rightarrow y\}, y\rangle$ is a rejected argument.

Example 2. *Consider the RP-DeLP program $\mathcal{P}_2 = (\Pi, \Delta, \preceq)$, with*

$$\Pi = \{\emptyset\}, \Delta = \{a, b, \sim c, a \wedge b \rightarrow c, b \rightarrow \sim b\}$$

and two defeasible sets as follows: $\{a \wedge b \rightarrow c, b \rightarrow \sim b\} \prec \{a, b, \sim c\}\{a \wedge b \rightarrow c, b \rightarrow \sim b\}$. Assume α_1 is the level of $\{a, b, \sim c\}$ and α_2 is the level of $\{a \wedge b \rightarrow c, b \rightarrow \sim b\}$, with $1 > \alpha_1 > \alpha_2 > 0$. As there is no information at strict level (Π), $\mathit{Warr}(1) = \emptyset$. Then at level α_1 there are three valid arguments: $\mathcal{A}_1 = \langle\{a\}, a\rangle$, $\mathcal{A}_2 = \langle\{b\}, b\rangle$ and $\mathcal{A}_3 = \langle\{\sim c\}, c\rangle$. As $\mathit{Warr}(> \alpha_1) \cup \{a, b, \sim c\} \not\vdash \bot$ then $\mathit{Warr}(\alpha_1) = \{a, b, \sim c\}$. In the next level $\mathcal{A}_4 = \langle\{b, a, a \wedge b \rightarrow c\}, c\rangle$ is a valid argument and as it minimally conflicts w.r.t $\mathit{Warr}(> \alpha_2)$ then $\mathit{Block}(\alpha_2) = \{c\}$.

In [ABG10] authors showed that, in case of some circular definitions of conflict among arguments, the output of an RP-DeLP program may be not unique, that is, there may exist several pairs (*Warr*, *Block*) satisfying the above conditions for a given RP-DeLP program.

The authors formalized circular definitions of conflicts by means of what they called *warrant dependency graphs*, following the approach of Pollock [Pol09]. To support the definition of the warrant dependency graphs the authors define the conditions that an argument must satisfy in order to be an *almost valid argument* wrt a set of valid arguments. This definition will be used later in the warrant dependency graph definition.

Let $\mathcal{P} = (\Pi, \Delta, \preceq)$ be an RP-DeLP program, let W and B be two sets of warranted and blocked conclusions, respectively, and let $\mathbb{A}$ be a set of valid arguments of strength α with respect to $W(\geq \alpha_i)$ and $B(> \alpha_i)$. An argument $\langle F, P\rangle$ of strength α is *almost valid* with respect to $\mathbb{A}$ if it satisfies the following six conditions:

(AV1) for any subargument $\langle C, R\rangle \sqsubset \langle F, P\rangle$ of strength $\beta > \alpha$, $R \in W(\beta)$;

(AV2) $P \notin W(> \alpha) \cup B(> \alpha)$;

(AV3) $\sim P \notin B(> \alpha)$ and $\Pi \cup W(> \alpha) \cup \{R \mid \langle C, R\rangle \sqsubset \langle F, P\rangle\} \cup \{P\} \not\vdash \bot$;

(AV4) there does not exist a valid argument for conclusion P of strength α;

(AV5) for any subargument $\langle C, R\rangle \sqsubset \langle F, P\rangle$ of strength α such that $R \notin W(\alpha)$, it holds that $\langle C, R\rangle \in \mathbb{A}$, otherwise R and $\sim R \notin B(\geq \alpha)$; and

(AV6) there exists at least an argument $\langle C, R\rangle \in \mathbb{A}$ such that $\langle C, R\rangle \sqsubset \langle F, P\rangle$.

Intuitively, an almost valid argument captures the idea of an argument based on valid arguments and which status is warranted (not rejected) whenever these subarguments are warranted, and rejected, otherwise. In particular, Condition (AV1) corresponds to a smoothed version of Condition (V1). Conditions (AV2) and (AV3) are equivalent to Conditions (V2) and (V3), respectively. Condition (V4) ensures that there does not exist a valid argument for the literal, and Conditions (AV5) and (AV6) ensure that the status of an almost valid argument depends on the status of at least one valid argument.

At this point we are ready to define the warrant dependency graph for a set of valid arguments and a set of almost valid arguments.

Definition 3.1 (Warrant dependency graph). *Let $\mathcal{P} = (\Pi, \Delta, \preceq)$ be an RP-DeLP program and let W and B be two sets of warranted and blocked conclusions, respectively. Moreover, let $\mathcal{A}_1 = \langle A_1, Q_1\rangle, \ldots, \mathcal{A}_k = \langle A_k, Q_k\rangle$ be valid arguments of strength α with respect to $W(\geq \alpha_i)$ and $B(> \alpha_i)$, and let $\mathcal{B}_1 = \langle B_1, P_1\rangle, \ldots, \mathcal{B}_n = \langle B_n, P_n\rangle$ be arguments of strength α that are almost valid with respect to $\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$. The warrant dependency graph (V, E) for $\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ and $\{\mathcal{B}_1, \ldots, \mathcal{B}_n\}$ is defined as follows:*

1. *For every literal $L \in \{Q_1, \ldots, Q_k\} \cup \{P_1, \ldots, P_n\}$, the set of vertices V contains one vertex v_L.*

2. *For every pair of literals $(L_1, L_2) \in \{Q_1, \ldots, Q_k\} \times \{P_1, \ldots, P_n\}$ such that the argument of L_1 is a subargument of the argument of L_2, the set of directed edges E includes one edge (v_{L_1}, v_{L_2}).*[4]

3. *For every pair of literals $(L_1, L_2) \in \{P_1, \ldots, P_n\} \times \{Q_1, \ldots, Q_k\}$ such that $L_1 = \sim L_2$, the set of directed edges E includes one edge (v_{L_1}, v_{L_2}).* [5]

4. *For every strict rule $R \leftarrow R_1 \wedge \ldots \wedge R_p \in \Pi$ such that $\{\sim R, R_1, \ldots, R_p\} \subseteq W(\geq \alpha) \cup \{Q_1, \ldots, Q_k\} \cup \{P_1, \ldots, P_n\}$, the set of directed edges E includes one edge (v_{L_1}, v_{L_2}) for every pair of literals $(L_1, L_2) \in \{P_1, \ldots, P_n\} \times \{Q_1, \ldots, Q_k\}$ such that the argument of L_2 is not a subargument of the argument of L_1, $L_1 \in \{\sim R, R_1, \ldots, R_p\}$ and, either $L_2 \in \{\sim R, R_1, \ldots, R_p\}$ or, L_2 is a subargument of the argument of L_3, for some $L_3 \in \{P_1, \ldots, P_n\}$ such that $L_3 \in \{\sim R, R_1, \ldots, R_p\}$.* [6]

5. *Elements of V and E are only obtained by applying the above construction rules.*

The idea behind the warrant dependency graph is that it represents

(i) support relations of almost valid arguments with respect to valid arguments and

(ii) conflict relations of valid arguments with respect to almost valid arguments.

Example 3. *Consider the RP-DeLP program $\mathcal{P}_3 = (\Pi, \Delta, \preceq)$, with*

$$\Pi = \{\emptyset\},\ \Delta = \{p, q, p \rightarrow \sim q, q \rightarrow \sim p\}.$$

Now, consider the empty set of conclusions $W = W(1) = \emptyset$ and arguments for conclusions p and q; i.e. $\mathcal{A}_1 = \langle\{p\}, p\rangle$ and $\mathcal{A}_2 = \langle\{q\}, q\rangle$. Finally, consider the arguments for conclusions $\sim p$ and $\sim q$; i.e.

$$\mathcal{B}_1 = \langle\{q,\ \sim p \leftarrow q\}, \sim p\rangle \text{ and}$$
$$\mathcal{B}_2 = \langle\{p,\ \sim q \leftarrow p\}, \sim q\rangle.$$

Figure 3.2 shows the warrant dependency graph for $\mathcal{A}_1$ and $\mathcal{A}_2$ w.r.t. $W = \emptyset$, $\mathcal{B}_1$, and $\mathcal{B}_2$. Conflict and support relationships among these arguments are represented as dashed and solid arrows, respectively. The cycle of the graph expresses that the warranty of p depends on the validity of $\sim p$, which depends on the warranty of q, which depends on the validity of $\sim q$, which in turn depends on the warranty of p.

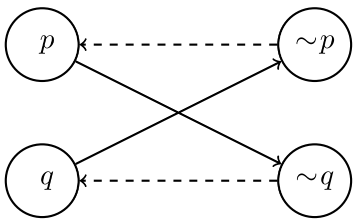


Figure 3.1: Warrant dependency graph for RP-DeLP program from Example 3

Example 4. *Consider the RP-DeLP program $\mathcal{P}_4 = (\Pi, \Delta, \preceq)$, with*

$$\begin{aligned}\Pi =& \{\emptyset\},\\ \Delta =& \{p, q, r, p \wedge q \to \sim r, r \wedge q \to \sim p, r \wedge p \to \sim q\}.\end{aligned}$$

As in the precedent example, consider the empty set of conclusions $W = W(1) = \emptyset$ and now we have arguments for conclusions p, q and r: $\mathcal{A}_1 = \langle\{p\}, p\rangle$, $\mathcal{A}_2 = \langle\{q\}, q\rangle$ and $\mathcal{A}_3 = \langle\{r\}, r\rangle$. Based on this valid arguments we have the following subarguments: $\mathcal{B}_1 = \langle\{p, q, p \wedge q \to \sim r\}, \sim r\rangle$, $\mathcal{B}_2 = \langle\{p, r, p \wedge r \to \sim q\}, \sim q\rangle$, $\mathcal{B}_3 = \langle\{q, r, q \wedge r \to \sim p\}, \sim p\rangle$.

Conflict and support relationships among these arguments are represented as dashed and solid arrows, respectively. The graph contains many cycles. For instance, following the recursive definition of valid and almost valid argument, the set of edges

$$\{(\sim p, p), (q, \sim p), (\sim q, q), (p, \sim q)\}$$

expresses that:

1. *The warranty of p depends on a (possible) conflict with $\sim p$ (direct conflict between p and $\sim p$ if $\sim p$ was valid).*
2. *The support of $\sim p$ depends on q (i.e. the validity of $\sim p$ depends on the warranty of q);*
3. *The warranty of q depends on a (possible) conflict with $\sim q$ (direct conflict between q and $\sim q$ if $\sim q$ was valid).*
4. *The support of $\sim q$ depends on p (i.e. the validity of $\sim q$ depends on the warranty of p).*

[4]The directed edge (v_{L_1}, v_{L_2}) represents an inference (subargument) relation from a valid argument to an almost valid argument.

[5]The directed edge (v_{L_1}, v_{L_2}) represents a direct conflict, inconsistency due to defeasible rules, between an almost valid argument and a valid argument.

[6]The directed edge (v_{L_1}, v_{L_2}) represents an indirect conflict, inconsistency due to strict rules, between an almost valid argument and a valid argument.

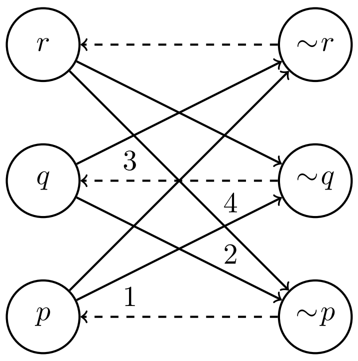


Figure 3.2: Warrant dependency graph for RP-DeLP program from example 4

The following example will show a circular definition of a cycle with strict knowledge.

Example 5. *Consider the RP-DeLP program* $\mathcal{P}_5 = (\Pi, \Delta, \preceq)$*, with*

$$\begin{aligned}\Pi = &\{\sim a, p_1 \land s \to a, q_1 \land r \to a\},\\ \Delta = &\{p_1, q_1, q_1 \to s, p_1 \to r,\\ &p_2, q_2, p_2 \to r, q_2 \to q_1,\\ &p_2 \to \sim q_2, q_2 \to s\}\end{aligned}$$

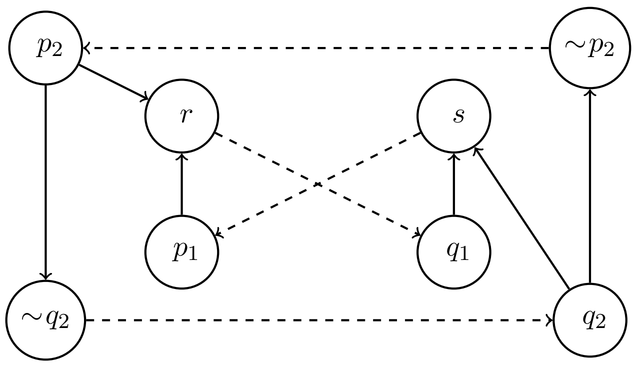


Figure 3.3: Warrant dependency graph for RP-DeLP program from example 5

In this example the strict knowledge (Π*) is not empty, so we have to consider* $W = W(1) = \{\sim a\}$ *and the arguments for* p_1, q_1, p_2 *and* q_2*:* $\mathcal{A}_1 = \langle\{p_1\}, p_1\rangle$*,* $\mathcal{A}_2 = \langle\{q_1\}, q_1\rangle$*,* $\mathcal{A}_3 = \langle\{p_2\}, p_2\rangle$ *and* $\mathcal{A}_4 = \langle\{q_2\}, q_2\rangle$*. Based on those valid arguments we have the following almost valid arguments:*

- *Almost valid arguments for r:* $\mathcal{B}_1 = \langle\{p_1, p_1 \to r\}, r\rangle$ *and* $\mathcal{B}_2 = \langle\{p_2, p_2 \to r\}, r\rangle$
- *Almost valid arguments for s:* $\mathcal{B}_3 = \langle\{q_1, q_1 \to s\}, s\rangle$ *and* $\mathcal{B}_4 = \langle\{q_2, q_2 \to s\}, s\rangle$
- *Almost valid argument for $\sim p_2$:* $\mathcal{B}_5 = \langle\{q_2, q_2 \to \sim p_2\}, \sim p_2\rangle$
- *Almost valid argument for $\sim q_2$:* $\mathcal{B}_6 = \langle\{p_2, p_2 \to \sim q_2\}, \sim q_2\rangle$

It is worth to notice that there are two different almost valid arguments for s and r. That can give a hint to see that despite of the fact that the number of nodes in the graph is upper bounded with the number of literals in the program, the number of almost valid arguments can be exponentially higher, so the complete contraction of the graph can take exponential time.

We can see that there are two cycles. The first one is

$$\{(s, p_1), (q_1, s), (r, q_1), (p_1, r)\},$$

and unlike the previous examples, there is strict knowledge involved in the circular definition of the conflict. There are two possible indirect conflicts at defeasible level: $\Pi \cup W \cup \{p_1, s\} \vdash \bot$ *and* $\Pi \cup W \cup \{q_1, r\} \vdash \bot$*. The complete cycle can be explained as follows:*

1. *The edge (s, p_1) indicates an (indirect) conflict between the valid argument for p_1 and an almost valid argument for s. That is, a conflict generated after considering the strict rules.*
2. *The edge (q_1, s) indicates that the almost valid argument for s depends on a valid argument for q_1.*
3. *The edge (r, q_1) indicates an (indirect) conflict between the valid argument for q_1 and an almost valid argument for r.*
4. *Finally, the edge (p_1, r) indicates that the almost valid argument for r depends on a valid argument for p_1.*

The second only has direct conflicts defined with the rules from the defeasible level

$$\{(\sim p_2, p_2), (q_2, \sim p_2), (\sim q_2, q_2), (p_2, \sim q_2)\},$$

and can be defined as follows:

1. *The edge $(\sim p_2, p_2)$ indicates a (direct) conflict between the valid argument for p_2 and an almost valid argument for $\sim p_2$.*
2. *The edge $(q_2, \sim p_2)$ indicates that the almost valid argument for $\sim p_2$ depends on a valid argument for q_2.*

3. *The edge ($\sim q_2, q_2$) indicates a (direct) conflict between the valid argument for q_2 and an almost valid argument for $\sim q_2$.*

4. *The edge ($p_2, \sim q_2$) indicates that the almost valid argument for $\sim q_2$ depends on a valid argument for p_2.*

Example 6. *Consider the RP-DeLP program $\mathcal{P}_6 = (\Pi, \Delta, \preceq)$, with*

$$\begin{aligned}\Pi =& \{p_1 \wedge s \to a, q_1 \wedge r \to a, p_2 \wedge s \to a, q_2 \wedge r \to a, \sim a\},\\ \Delta =& \{p_1, q_1, q_1 \to s, p_1 \to r, p_2, q_2, q_2 \to s, p_2 \to r\}.\end{aligned}$$

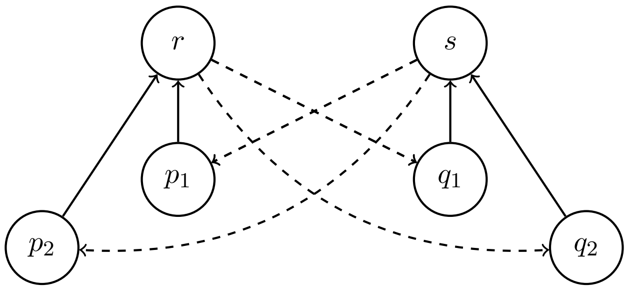


Figure 3.4: Warrant dependency graph for RP-DeLP program from example 6

In this example we have four arguments $\mathcal{A}_1 = \langle\{p_1\}, p_1\rangle$, $\mathcal{A}_2 = \langle\{p_2\}, p_2\rangle$, $\mathcal{A}_3 = \langle\{q_1\}, q_1\rangle$ and $\mathcal{A}_4 = \langle\{q_2\}, q_2\rangle$ such that any of them appears in two cycles in the dependency graph. Each cycle is associated to a different explanation of why p_1, p_2, q_1 and q_2 cannot be uniquely warranted or blocked, so in principle detecting any of the two cycles is enough to show that they are not uniquely determined.

For example, the cycle:

$$\{(s, p_1), (q_1, s), (r, q_1), (p_1, r)\}$$

Shows a circular definition between literals because:

1. *The edge (s, p_1) indicates an (indirect) conflict between the valid argument for p_1 and an almost valid argument for s. That is, a conflict generated after considering the strict rules.*

2. *The edge (q_1, s) indicates that the almost valid argument for s depends on a valid argument for q_1*

3. *The edge (r, q_1) indicates an (indirect) conflict between the valid argument for q_1 and an almost valid argument for r.*

4. *Finally, the edge* (p_1, r) *indicates that the almost valid argument for* r *depends on a valid argument for* p_1

That is, the status of warranted or blocked for the literals with valid arguments p_1 *and* q_1 *depends on each other, so our recursive based semantics does not give an unique status for these literals*

But for showing that p_1 *cannot be uniquely determined, we have also this other cycle:*

$$\{(s, p_1), (q_2, s), (r, q_2), (p_1, r)\}$$

It basically differs with the previous cycle in the almost valid argument for s*, that is now an argument that depends on* q_2 *instead of on* q_1 *So, with this cycle we discover that the status of* p_1 *and* q_1 *depends on each other. But observe that for certifying that* p_1 *cannot be uniquely determined to be warranted or blocked, any one of these two cycles is equally good*

We have a similar situation for literals p_2 *and* q_2.

In the next chapter we will present an algorithm to compute all the possible outputs from an RP-DeLP program. After presenting a first version of the algorithm which uses the dependency graph, we will also present an optimized version which avoids using the graph. This optimized algorithm is able to determine when a literal is part of a cycle avoiding explicitly working with the full constructed dependency graph. That optimization permits a better memory management as its not necessary to maintain a complete list of arguments for a literal.

4 On the Implementation of the RP-DeLP Framework

In this chapter we propose an algorithm which computes all possible RP-DeLP program outputs. Once a first version of the algorithm is presented, we propose an optimization which demonstrates that it is not necessary to construct the whole Warrant Dependency Graph to compute the outputs of an RP-DeLP program. Once this optimization is showed, we present an optimized algorithm. The complexity of this new optimized algorithm relies on two combinatorial queries. We also propose two encodings for solving such queries: one based in SAT techniques and the other one based on ASP techniques.

Contents

4.1. Computing the Set of Outputs for an RP-DeLP Program

From a computational point of view, the set of outputs for a recursive based semantics can be computed by means of a recursive algorithm, starting with the computation of warranted conclusions from strict clauses and recursively going from warranted conclusions to defeasible arguments based on them.

For every level the algorithm must compute the sets of valid and almost valid arguments and has to check the existence of conflicts between valid arguments and cycles at some warrant dependency graph.

As we have seen in some examples in the precedent chapter, the existence of cycles leads to different outputs which satisfy the conditions of the RP-DeLP semantics. For a complete computation of all possible outputs, when a cycle is found one must exhaustively compute all the possible extensions resolving the cycle by adding one literal from the cycle into the set of warranted or blocked conclusions. To store the set of partially computed extensions at every level we use a stack.

In the following we use the simpler notation W, $W(1)$, $W(\alpha)$ and $W(\geq \alpha)$ for *Warr*, *Warr*(1), *Warr*(α) and *Warr*($\geq \alpha$) respectively, and B, $B(\alpha)$ and $B(\geq \alpha)$ for *Block*, *Block*(α) and *Block*($\geq \alpha$), respectively.

Algorithm 1.

```
Algorithm RP-DeLP_outputs
Input $\mathcal{P} = (\Pi, \Delta, \preceq)$: An RP-DeLP program
Output $O$: Set of outputs for $\mathcal{P}$
Variables
    $(W, B)$: Current output for $\mathcal{P}$
    $S$: Stack of partially computed outputs $(W, B, \alpha)$ for $\mathcal{P}$
Method
    $O := \emptyset$;
    $W := \{Q \mid \Pi \vdash Q\}$;
    $B := \emptyset$;
    $\alpha := 1$;
    Push$(S, (W, B, \alpha))$;
    while $(\neg$Empty_Stack$(S))$ do
        $(W, B, \alpha) :=$ Pop$(S)$;
        while $(\alpha >$lowest_level$(\preceq))$ do
            $\alpha :=$ next_level$(\preceq, \alpha)$;
            level_computing$(W, B, \alpha, S, O)$
        end while
    end while
end algorithm
```
Algorithm 1: RP-DeLP_outputs

The algorithm `RP-DeLP_outputs` first computes the set of warranted conclusions form the set of strict clauses Π and uses the stack S to store the set of partially computed outputs for each level. We define function `Pop(S)`, which returns and deletes the top of the stack S. `Empty_Stack(S)` returns `True` if the stack S is empty and `False` otherwise.

Elements in S are (W, B, α) which represent partially computed outputs where W and B are warranted and blocked conclusions up to level α. For example, imagine we have a program $\mathcal{P}$ with n defeasible levels and at level l exists one cycle involving two literals: u and v. The algorithm detects the cycle at level l which implies that there are at least two different outputs satisfying the conditions of RP-DeLP semantics. After the execution of `level_computing` at level l the stack S will contain two elements: one for the extension for v at level i and the other for the extension for u at level i. Those two extensions will be extended at the following levels until the lower level, where they will be added at set O.

When all possible extensions of a partial output are explored in the lowest defeasible level, variable O stores the output. Take into account that, at the lowest level computed outputs will not be stored in S as long they are not partial outputs. All outputs at lowest level will be stored at O which is the set representing complete outputs. Then, as long as there are elements in S, there are partial outputs unexplored. Algorithm `RP-DeLP_outputs` will not finish until all the possible outputs are explored.

For every partially computed output and every defeasible level $1 > \alpha > 0$, the procedure `level_computing` extends the partially computed outputs with the sets of warranted and blocked conclusions of the next level α. Function `level_computing` returns all the extensions up to level α. Function `lowest_level` returns the lowest defeasible level in program $\mathcal{P}$, and function `next_level` returns iteratively in descendant order the following value of α.

Algorithm `RP-DeLP_outputs` finishes when S is empty and O contains all the possible outputs of $\mathcal{P}$.

```
Procedure level_computing (in_out $W$, $B$, $\alpha$, $S$, $O$)
Variable $VA$: Set of valid arguments
Method
    $VA := \{\langle A, Q\rangle$ with strength $\alpha \mid \langle A, Q\rangle$ is valid w.r.t. $W$ and $B\}$;
    weighted_extension ($W$, $B$, $\alpha$, $VA$, $S$, $O$);
    $(W, B, \alpha) :=$ Pop($S$);
end procedure level_computing
```

The procedure `level_computing` receives a partially computed output with a set of warranted conclusions $W(> \alpha)$ and a set of blocked conclusions $B(> \alpha)$, and extends this partial output up to level α, i.e. computes $W(\geq \alpha)$ and $B(\geq \alpha)$ for every extension. As `level_computing` computes the partial output of the next level of α, first of all we have to compute the set of valid arguments in α.

```
Procedure weighted_extension (in $W$, $B$, $\alpha$, $VA$; in_out $S$, $O$)
Variables
    $W_{ext}$: Extended set of warranted conclusions
    $VA_{ext}$: Extended set of valid arguments
    $is\_leaf$: Boolean
Method
```

```
        $is\_leaf$ := true;
        **while** ($VA \neq \emptyset$ **and** $is\_leaf$ = true) **do**
            **while** ($\exists \langle A, Q\rangle \in VA$ |
            $\neg$ conflict($\langle A, Q\rangle$, $VA$, $W$, not_dependent($\langle A, Q\rangle$,
            almost_valid($VA$, ($W$, $B$))))
            **and** $\neg$ cycle($\langle A, Q\rangle$, $VA$, $W$, almost_valid($VA$, ($W$, $B$))) **do**
                    $W := W \cup \{Q\}$;
                    $VA := VA \backslash \{\langle A, Q\rangle\} \cup \{\langle E, P\rangle$ with strength $\alpha \mid \langle E, P\rangle$
                    is valid w.r.t. $W$ and $B\}$
            **end while**
            $I := \{\langle A, Q\rangle \in VA \mid$ conflict($\alpha$, $\langle A, Q\rangle$, $VA$, $W$, $\emptyset$) $\}$;
            $B := B \cup \{Q \mid \langle A, Q\rangle \in I\}$;
            $VA := VA \backslash I$;
            $J := \{\langle A, Q\rangle \in VA \mid$ cycle($\alpha$, $\langle A, Q\rangle$, $VA$, $W$,
            almost_valid($\alpha$, $VA$, $W$, $B$)) $\}$;
            **for each argument** ($\langle A, Q\rangle \in J$) **do**
                $W_{ext} := W \cup \{Q\}$;
                $VA_{ext} := VA \backslash \{\langle A, Q\rangle\} \cup \{\langle E, P\rangle$ with strength $\alpha \mid \langle E, P\rangle$
                is valid w.r.t. $W_{ext}$ and $B\}$;
                weighted_extension ($W_{ext}$, $B$, $\alpha$, $VA_{ext}$, $S$, $O$)
            **end for**
            **if** ($J \neq \emptyset$) **then** $is\_leaf$ := false
        **end while**
        **if** (($W, B, \alpha) \notin S$ **and** $is\_leaf$ = true ) **then**
                Push($S$, ($W, B, \alpha$));
                **if** ($\alpha$ = lower_level($\preceq$)) **then** $O := O \cup \{(W, B)\}$
        **end if**
**end procedure** weighted_extension
```

The recursive procedure `weighted_extension` receives as input a partially computed output (W, B) at a level α and the set of valid arguments VA with respect to W and B. It also receives S and O as input/output variables, which will only be used to store the partial or complete output respectively.

When a cycle is found in a warrant dependence graph, each valid argument of the cycle can lead to a different output. Then, the procedure `weighted_extension` selects one valid argument of the cycle and recursively computes the resulting output by warranting the selected argument. The procedure finishes when the status for every valid argument is computed. If the recursive analysis leads to a new extension, it is stored in the stack S. Moreover, if the level computing α corresponds with the lower strength of the program arguments, each new output is added to the set of outputs O.

The function `almost_valid` computes the set of almost valid arguments based on some valid arguments in VA. The function `not_dependent` computes the set of almost valid arguments which do not depend on $\langle A, Q\rangle$.

The function `conflict` has two different functionalities:

- On the one hand, the function `conflict` checks possible conflicts among the argument $\langle A, Q\rangle$ and the set VA of valid arguments extended with the set of almost valid arguments whose supports depend on some argument in $VA\backslash\{\langle A,Q\rangle\}$, and thus, every valid argument with options to be involved in a conflict remains as valid.

$$\texttt{conflict}(\langle A,Q\rangle, VA, W, \texttt{not\_dependent}(\langle A,Q\rangle, \texttt{almost\_valid}(VA,(W,B))))$$

- On the other hand, the function `conflict` checks conflicts among the argument $\langle A,Q\rangle$ and the set VA of valid arguments, and thus, every valid argument involved in a conflict is blocked.

$$\texttt{conflict}(\langle A,Q\rangle, VA, W, \emptyset)$$

Finally, the function `cycle` checks the existence of a cycle in the warrant dependency graph for the set of valid arguments VA and the set of almost valid arguments based on some valid arguments in VA. Please note that the function `cycle` needs the full constructed dependency graph in order to check if a literal is involved in a conflict. Despite of the fact that it can be showed that once the graph is constructed, checking that a literal is involved in a cycle can be solved in polynomial time, there is still the need of the construction of the graph which can take exponential time with respect to the number of literals in the program.

It can be we showed that whenever `cycle` returns true for $\langle A,Q\rangle$, then a conflict will be detected with the set of almost valid arguments which do not depend on $\langle A,Q\rangle$. Moreover, the set of valid arguments J computed by function `cycle` can also be computed by checking the stability of the set of valid arguments after two consecutive iterations, so it is not necessary to explicitly compute dependency graphs.

Proposition 4.1 (Optimization). *Let $\mathcal{P} = (\Pi, \Delta, \preceq)$ be an RP-DeLP program with defeasibility levels $1 > \alpha_1 > \ldots > \alpha_p > 0$ for Δ, and let W and B be two sets of warranted and blocked conclusions with strength $\geq \alpha_i$, respectively. If VA is the set of all d-arguments of strength α_i that are valid with respect to (W,B) and AV is the set of all d-arguments of strength α_i that are almost valid with respect to VA, we get the following results:*

(i) If there is a cycle in the warrant dependence graph for VA and AV, and $\langle A,Q\rangle \in VA$ is such that the vertex of conclusion Q is a vertex of the cycle or there exists a path from a vertex of the cycle to the the vertex of conclusion Q, then there exists a set $ND \subseteq AV$ such that $\langle A,Q\rangle \not\sqsubseteq \langle R,P\rangle$ for all $\langle R,P\rangle \in ND$, and there exists a set $S \subseteq VA\backslash\{\langle A,Q\rangle\}$ such that $\Pi \cup W \cup \{P \mid \langle B,P\rangle \in S\} \cup ND \not\vdash \bot$ and $\Pi \cup W \cup \{P \mid \langle B,P\rangle \in S\} \cup ND \cup \{Q\} \vdash \bot$.

(ii) If for all $\langle A,Q\rangle \in VA$ there exists a set $ND \subseteq AV$ such that $\langle A,Q\rangle \not\sqsubseteq \langle R,P\rangle$, for all $\langle R,P\rangle \in ND$, and there exists a set $S \subseteq VA\backslash\{\langle A,Q\rangle\}$

such that $\Pi \cup W \cup \{P \mid \langle B, P\rangle \in S\} \cup ND \not\vdash \bot$ *and* $\Pi \cup W \cup \{P \mid \langle B, P\rangle \in S\} \cup ND \cup \{Q\} \vdash \bot$*, then there is at least a cycle in the warrant dependence graph for VA and AV, and every* $\langle A, Q\rangle \in VA$ *is such that the vertex of conclusion* Q *is a vertex of a cycle or there exists a path from a vertex of a cycle to the the vertex of conclusion* Q*.*

Proof.

(i) If the vertex of conclusion Q is a vertex of the cycle, because of the warranty dependency graph definition, we can consider the set $ND \subseteq AV$ such that the vertex of each conclusion in ND is a vertex of the cycle and $\langle A, Q\rangle \not\sqsubset \langle R, P\rangle$ for all $\langle R, P\rangle \in ND$, and then, there should exist a set $S \subseteq VA \backslash \{\langle A, Q\rangle\}$ such that $\Pi \cup W \cup \{P \mid \langle B, P\rangle \in S\} \cup ND \not\vdash \bot$ and $\Pi \cup W \cup \{P \mid \langle B, P\rangle \in S\} \cup ND \cup \{Q\} \vdash \bot$. If the vertex of conclusion Q is not a vertex of the cycle and there exists a path from a vertex of the cycle to the the vertex of conclusion Q, we can consider the set $ND \subseteq AV$ such that the vertex of each conclusion in ND is a vertex of the cycle. Now, because of the warranty dependency graph definition, $\langle A, Q\rangle \not\sqsubset \langle R, P\rangle$ for all $\langle R, P\rangle \in ND$ and there should exist a set $S \subseteq VA \backslash \{\langle A, Q\rangle\}$ such that $\Pi \cup W \cup \{P \mid \langle B, P\rangle \in S\} \cup ND \not\vdash \bot$ and $\Pi \cup W \cup \{P \mid \langle B, P\rangle \in S\} \cup ND \cup \{Q\} \vdash \bot$.

(ii) We have that for all $S \subseteq VA$, $\Pi \cup W \cup \{P \mid \langle R, P\rangle \in S\} \not\vdash \bot$ and that for all $\langle A, Q\rangle \in VA$ there exists a set $ND \subseteq AV$ such that $\langle A, Q\rangle \not\sqsubset \langle R, P\rangle$ for all $\langle R, P\rangle \in ND$, and there exists a set $S \subseteq VA \backslash \{\langle A, Q\rangle\}$ such that $\Pi \cup W \cup \{P \mid \langle B, P\rangle \in S\} \cup ND \not\vdash \bot$ and $\Pi \cup W \cup \{P \mid \langle B, P\rangle \in S\} \cup ND \cup \{Q\} \vdash \bot$. Then, for all $\langle A, Q\rangle \in VA$, we have that the warranty of Q depends on a possible conflict with a set $S \subseteq VA \backslash \{\langle A, Q\rangle\}$ and a set $ND \subseteq AV$ such that $\langle A, Q\rangle \not\sqsubset \langle R, P\rangle$ for all $\langle R, P\rangle \in ND$. Therefore, because of the warranty dependency graph definition, there should exists a cycle in the warrant dependence graph (V, E) for VA and AV such that the vertexes of conclusions of ND are vertexes of the cycle and the vertexes of conclusions of S and $\{\langle A, Q\rangle\}$ are vertexes of the cycle or there exists a path from a vertex of the cycle to the the vertex of these conclusions.

∎

From Proposition 4.1 it follows that we only need one argument per conclusion in order to find a conflict among arguments. So, there is no need for maintaining the full list of arguments for a conclusion.

Next we present an optimized version of the previously presented algorithm. This new version uses the same main algorithm `RP-DeLP_output` using two new versions of procedures `level_computing` and `weighted_extension`.

The new optimized version of the algorithm makes use of a new set of literals. This new set is called *VL* and it contains the set of *valid literals.* A valid literal is the conclusion of a valid argument (similarly we will refer as *almost valid literals* to the conclusions of almost valid arguments).

```
Procedure level_computing (in_out $W$, $B$, $\alpha$, $S$, $O$)
Variable $VL$: Set of valid literals
Method
    $VL := \{Q$ with strength $\alpha \mid \langle A, Q\rangle$ is valid w.r.t. $W$ and $B\}$;
    weighted_extension ($W$, $B$, $\alpha$, $VL$, $S$, $O$);
    $(W, B, \alpha) :=$ Pop($S$);
end procedure level_computing

Procedure weighted_extension (in $W$, $B$, $\alpha$, $VL$; in_out $S$, $O$)
Variables
    $W_{ext}$: Extended set of warranted conclusions
    $VL_{ext}$: Extended set of valid literals
    $is\_leaf$: Boolean
Method
    $is\_leaf :=$ true;
    while ( $VL \neq \emptyset$ and $is\_leaf =$ true) do
        while ($\exists Q \in VL \mid$
        $\neg$ conflict($\alpha$, $Q$, $VL$, $W$, almost_valid_literals($\alpha$, $VL$, ($W$, $B$), $Q$))) do
                $W := W \cup \{Q\}$;
                $VL := VL \setminus \{Q\} \cup \{P$ with strength $\alpha \mid \langle E, P\rangle$
                is valid w.r.t. $W$ and $B\}$
        end while
        $I := \{Q \in VL \mid$ conflict($\alpha$, $Q$, $VL$, $W$, $\emptyset$) $\}$;
        $B := B \cup I$;
        $VL := VL \setminus I$;
        if $VL$ does not change from previous iteration then $J := VL$;
        for each literal ($Q \in J$) do
            $W_{ext} := W \cup \{Q\}$;
            $VL_{ext} := VL \setminus \{Q\} \cup \{P$ with strength $\alpha \mid \langle E, P\rangle$
            is valid w.r.t. $W_{ext}$ and $B\}$;
            weighted_extension ($W_{ext}$, $B$, $\alpha$, $VL_{ext}$, $S$, $O$)
        end for
        if ($J \neq \emptyset$) then $is\_leaf :=$ false
    end while
    if (($W, B, \alpha) \notin S$ and $is\_leaf =$ true ) then
            Push($S$, ($W, B, \alpha$));
            if ($\alpha =$ lowest_level($\preceq$)) then $O := O \cup \{(W, B)\}$
    end if
end procedure weighted_extension
```

The main difference introduced in `level_computing` respect to the procedure in the original algorithm is that it makes use of a list of valid literals as input instead of a list of arguments.

So now, in the optimized version, `weighted_extension` procedure will take a list of literals instead of a list of arguments. Another difference in `weighted_extension` is the absence of the function `cycle` as long as is not necessary to find cycles in the warrant dependency graph as follows from Proposition 4.1. Instead of `cycle` function, we check the stability of the set of valid literals in two iterations. We can check that if in two iterations of the main loop in `weighted_extension` procedure the set of VL has not changed, all literals in set VL are involved in at least a cycle with other literals in VL. Nevertheless we can not obtain the full description of the cycle, so by checking the stability of set VL we can assure that there is at least one cycle and also that all literals in set VL are in a cyclic definition of a conflict with other valid and almost valid literals.

Working this way we can detect the existence of cyclic definitions of conflicts but without explicitly computing them. As a drawback we are not capable to give a full description of the cycle, but we can speed up and optimize the algorithm.

Similarly as in the non optimized version, for any level α the procedure `weighted_extension` first computes the set VL of valid literals with respect to $W(> \alpha)$ and $B(> \alpha)$. Then, this set of valid literals is dynamically updated depending on new warranted and blocked conclusions with strength α. To do so, we maintain a list with the rules which can derive new valid arguments. When a literal is added to W, then this list is used to quickly check for new updates. Given the recursive definition of a valid argument, its conditions can be checked in polynomial time. Then it only remains assuring that new literals do not conflict with strict knowledge. Also, when a literal is added to B, we update the list and delete the rules containing this new blocked literal in its body. The procedure `weighted_extension` finishes when the status for every valid literals is resolved.

Hence the `weighted_extension` procedure needs to compute two main queries during its execution: i) to find the set of almost valid literals not depending on valid literal Q, ii) check if there is a conflict for a valid literal given a set of valid literals and optionally a set of almost valid literals.

In the following we present SAT and ASP encodings for these two main combinatorial queries. The input and output specification of each query is as follows:

(i) `almost_valid_literals`(α, VL, (W, B), Q): It takes as input a set VL of valid literals of strength α, sets W and B of warranted and blocked literals of strength $\geq \alpha$, respectively, and a literal Q such as the argument $\langle A, Q\rangle$ is a valid argument with respect to W and B at level α. It iteratively checks among the set of possible almost valid literals for the existence of an almost valid argument of strength α that does not depend on Q. The query will return a list of almost valid literals with at most one argument for each literal.

Note that the status of an almost valid argument $\mathcal{B}$ only changes under the following conditions:

i When all subarguments of $\mathcal{B}$ are in the set W, then the almost valid argument changes its status to valid argument.

ii When one subargument of $\mathcal{B}$ is in the set B, it is no longer almost valid as it will not satisfy the conditions of a valid argument.

For each almost valid literal we maintain a cache with the last call to function `almost_valid_literal` where the status of an almost valid literal was checked. So, before launching the query we can check this cache. If we find a previous recorded entry in this cache, it means that the status of literal has not changed. If we do not find an entry, then we launch the query. There is a necessary task to do, and it is to keep the cache updated, so we have to detect when an almost valid argument changes its status due to the inclusion of a literal in the set of the warranted or blocked.

One can see that finding an almost valid argument for a literal is a combinatorial problem. This is the problem of finding a set of almost valid literals not depending on Q and to solve such problem we make use of SAT and ASP techniques. In next section we present two encodings based on SAT and ASP designed for efficiently solve those tasks. Moreover, using those encodings we can find an almost valid argument not depending on a literal Q.

(ii) `conflict`(α, Q, VL, W, ND): It takes as input a set W of warranted literals of strength $\geq \alpha$, a set VL of valid literals of strength α, a valid literal Q of strength α, and a set ND of almost valid literals of strength α that do not depend on Q. It checks (possible) conflicts among the literal Q and the set VL of valid arguments extended with the set ND of almost valid arguments.

Observe that there are two `conflict` calls. The first one with a non empty set of ND almost valid literals which checks for a possible conflict between literal Q and $W \cup VL \cup ND$. The importance of this call is that if no conflict is found we set literal Q as warranted as Q will not be in any conflict.

In the second query we have $ND = \emptyset$, so if a conflict is found, it means that there is at least a conflict between Q and $W \cup VL$, so conclusion Q must be blocked. Finding conflicting arguments for conclusion Q among the rules and facts from an $RP - DeLP$ program is a combinatorial problem. Again, to solve such problem we make use of ASP and SAT techniques.

In the next section we also present two encodings based on SAT and ASP techniques which solve such problem in an effective way. Moreover SAT and ASP queries will return a subset $C \subseteq W \cup VL$ which is not minimum but satisfies $Q \cup C \cup \Pi \vdash \perp$. We can use the set of literals C to give an explanation of why we block the conclusion Q.

We can see that this algorithm will compute an exponential number of outputs respect to the size of the program in worst case. For each output it will perform linear number of calls to the solvers.

4.2. SAT Encodings for Finding Warranted Literals

In the previous section we have seen that we can compute the outputs of an RP-DeLP program by means of a level-wise procedure, starting from the highest level and iteratively going down from one level to next level below. At every level it is necessary to determine the status (warranted or blocked) of each valid argument by checking the existence of both conflicts between arguments, and cycles at the warrant dependence graphs. We also showed that this level-wise procedure can be implemented to work in polynomial space. On the one hand this can be achieved because it is not actually necessary to find all the valid arguments for a given literal, it is enough to find only one. Actually, in our implementation to explain the existence of a valid argument for a literal Q we simply record the *last* rule of the argument, that is, a rule with Q as conclusion, and with all the literals of its body as warrants. To give a full explanation for a valid argument, we recursively give explanations for all the warrants of the body of the rule. Something similar applies to the computation of at most one almost valid argument for a given literal. This will be done with the first of the two SAT encodings we present next, and it allows also to explicitly give an almost valid argument for a literal, not only to check the existence. On the other hand, the existence of cycles in the warrant dependency graph among valid and almost valid arguments can be validated by checking the stability of conflicts between valid and almost valid arguments, so it is not necessary to explicitly compute the warrant dependency graphs. Hence, the procedure to find warranted literals needs to compute two main queries during its execution:

i whether an argument is almost valid, and

ii whether there is a conflict among valid and almost valid arguments.

4.2.1. Looking for Almost Valid Arguments

The idea for encoding the problem of searching almost valid arguments is based on the same behind successful SAT encodings for solving STRIPS planning problems [KS99b]. In a STRIPS planning problem, given an initial state, described with a set of predicates, the goal is to decide whether a desired goal state can be achieved by means of the application of a suitable sequence of actions. Each action has a set preconditions, when they hold true the action can be executed and as a result certain facts become true and some others become false (its effects). Hence executing an action changes the current state, and the application of a sequence of actions creates a sequence of states. The planning problem is to find a sequence of actions such that, when executed, the obtained final state satisfies the goal state.

In our case, the search for an almost valid argument $\langle C, P\rangle$ can be seen as the search for a *plan* for producing P, taking as the initial set of facts some subset of a set of literals in which we already trust. We call such initial set the base set of literals[1], and we say that they are true at the first step of the argument. For looking for an almost valid argument $\langle C, P\rangle$, we will consider what rules should be executed, such that starting from the initial set will finally obtain the desired goal P. We say that a rule R can be executed starting from a set of literals S, when $\mathit{Body}(R) \subseteq S$, and that when it is executed we obtain a new set $S \cup \{\mathit{Head}(R)\}$. We have to consider only some rules for looking for almost valid arguments of strength α for literals not yet warranted, we called those rules α-rules and we say a rule R is an α-rule if it satisfies the following conditions:

1. Either $s(R) > \alpha$ and $\mathit{Body}(R) \setminus W(> \alpha) \neq \emptyset$, or $s(R) = \alpha$.
2. $\mathit{Body}(R) \cap B(\geq \alpha) = \emptyset$.
3. $\mathit{Head}(R), \sim\mathit{Head}(R) \notin W(\geq \alpha) \cup B(\geq \alpha)$.
4. There is no $\langle C, \mathit{Head}(R)\rangle \in \mathit{VA}$.

We use the following sets of literals and rules to define our SAT encoding. Consider first the initial set S_0:

$$S_0 = \{L \mid L \in W(\geq \alpha) \text{ or } \exists\langle C, L\rangle \in VA\}$$

which is the base set of warranted and valid literals. If we execute *all* the $\alpha-$rules that can be executed from S_0, that is:

$$R_0 = \{R \mid R \in \alpha\text{−rules}\ ,\ \mathit{Body}(R) \subseteq S_0\}$$

we obtain a new state S_1 that contains S_0 plus the heads of all the executed rules. This process can be repeated iteratively, obtaining a sequence of sets of literals $\mathcal{S} = \{S_0, S_1, \ldots, S_t\}$ and a sequence of sets of executed rules $\mathcal{R} = \{R_0, R_1, \ldots, R_{t-1}\}$, until we reach a final state S_t in which the execution of any possible rule does not increase the set of literals already in S_t. If starting from an initial set S_0 that contains all the current valid and warranted literals

[1]For an almost valid argument, the base set can contain only warranted and valid literals.

the final state S_t contains P, that means that an almost valid argument for P could be obtained from the sequence of executed rules, if we could find a subset of rules such that they can form an argument that satisfies all the conditions for an almost valid argument for P.

Observe that an almost valid argument $\langle C, P \rangle$ with strength α can only exist if the following conditions, that can be checked in polynomial time, are satisfied:

1. $P \notin W(> \alpha) \cup B(> \alpha)$. This is actually condition (AV2).
2. $\sim P \notin B(> \alpha)$. This is actually the first part of condition (AV3).
3. There does not exist a valid d-argument for conclusion P of strength α. This is actually condition (AV4).
4. $P \in S_t$.

If the previous conditions are satisfied, we proceed the search for $\langle C, P \rangle$ with strength α with a SAT encoding from the sequences $\mathcal{S}$ and $\mathcal{R}$ defined above.

That is, a SAT instance with variables to represent all the possible literals we can select from each set S_i:

$$\{v_L^i \mid L \in S_i, 0 \leq i \leq t\}$$

plus variables to represent all the possible rules R we can select from each set R_i:

$$\{v_R^i \mid R \in R_i, 0 \leq i < t\}$$

In order to check that the variables set to true represent an almost valid argument, we add clauses for ensuring that:

(1) If variable v_L^i is true, then either v_L^{i-1} is true or one of the variables v_R^{i-1}, with $\mathit{Head}(R) = L$, is true.

(2) If a variable v_R^i is true, then for all the literals L in its body v_L^i must be true.

(3) If variable v_L^i is true, then v_L^{i+1} is also true.

(4) The variable v_P^t must be true.

(5) No two contradictory variables v_L^t and $v_{\sim L}^t$ can be both true.

In addition, in order to satisfy the consistency of the literals of the argument with respect to the closure of the strict knowledge Π, we create also an additional set of variables V_Π and set of clauses R_Π. The set of variables V_Π contains a variable v_L^Π for each literal that appears in the logical closure of the set $S_t \cup W$ with respect to the strict rules.

Then, we add the following clauses to check the consistency with Π:

(1) If a literal is selected for the argument (v^t_L set to true) then v^Π_L must also be true.

(2) For any $L \in W$, v^Π_L must be true.

(3) For any rule $R \in \Pi$ that was executed when computing the logical closure, if for all the literals L in its body v^Π_L is true, then $v^\Pi_{Head(R)}$ must be true.

(4) No two contradictory variables v^Π_L and $v^\Pi_{\sim L}$ can be both true.

Observe that this *layered* encoding for searching almost valid arguments allows to explicitly recover the full structure of the argument, because we have both the literals and the rules that have generated them at each step of the argument.

We next show that any solution for a formula obtained with this SAT encoding gives an almost valid argument $\langle C, P\rangle$; i.e. we show that $\langle C, P\rangle$ satisfy Conditions (AV1)-(AV6):

(AV1) Given that the only possible rules that can be selected for building the almost valid argument are those defined as α-rules, the conclusion of an α-rule can only become valid with strength at most α. So, the only possible subarguments with strength greater than α are the ones corresponding to literals that can selected from the set S_0. Observe that the literals in the set S_0 are all the literals at the current set $W(\geq \alpha)$ plus the current set of literals with valid arguments with strength α. It follows that the only subarguments of strength $\beta > \alpha$ that can be implicitly used are the ones corresponding to warranted literals.

(AV2) This condition is actually checked before creating the SAT encoding. That is, if the condition is not satisfied, we answer that there is no such almost valid argument.

(AV3) The first part of this condition $\sim P \notin B(> \alpha)$ is also checked before creating the SAT encoding. For the second part, first observe that for any subargument $\langle C, R\rangle \sqsubset \langle B, P\rangle$, v^t_R will be true, due to the clauses in $(A3)$, as long as v^t_P (due to the clauses in $(A4)$). Then the clauses in $(B1)$ ensure that for any literal L with v^t_L true, the corresponding variable v^Π_L of the second part of the encoding will be also true. Finally, the clauses in $(B2)$, $(B3)$ and $(B4)$ will ensure that all such true literals are consistent with $\Pi \cup W$.

(AV4) As in the condition $(AV2)$, this condition is checked before creating the SAT encoding.

(AV5) Any literal L that is part of the argument ($v_L^t = true$) will be either generated by an α-rule, so it holds that $L, \sim L \notin W(\geq \alpha) \cup B(\geq \alpha) \cup VA$, or it is already true at the initial set ($v_L^0 = true$), so it is warranted or valid.

(AV6) Observe that there is no $R \in \alpha$-rules such that $Head(R) \in B \cup W \cup VA$, then it cannot be that all the rules used in the argument for P depend only on warranted literals from S_0 because that would mean that P is indeed valid (so P would have to be in VA). So, from the initial set S_0 at least one valid, but not warranted, literal will be activated, if any almost valid argument for P exists.

4.2.2. Looking for Collective Conflicts

We reduce the query computed by function `conflict`, to a query where we consider finding the set of conflict literals that are the conclusions of the corresponding conflict set of arguments. Basically, for finding this conflict set of literals S for a valid argument $\langle A, Q \rangle$ from the base set of literals considered in function `conflict`, i.e. the set $G = \{P \mid \langle C, P \rangle \in VA \setminus \{\langle A, Q \rangle\} \cup ND\}$, we have to find two arguments $\langle A_1, L \rangle$, $\langle A_2, \sim L \rangle$ using only rules from Π, literals $W \cup \{Q\}$ and a subset S from G, but such that when Q is not used, no conflict (generation of L and $\sim L$ for any L with strict rules) is produced with such set S. So, this can be seen as a simple extension of the previous query, where now we have to look for two arguments, instead of only one, although both arguments must be for two contradictory literals. That is, the SAT formula contains variables for encoding arguments that use as base literals $W \cup G \cup \{Q\}$ and rules from Π (with the same scheme of the previous SAT encoding for almost valid arguments), with an additional set of conflict variables to encode the set of possible conflicts that can be, potentially, generated from $W \cup G \cup \{Q\}$ using rules from Π, in order to be able to force the existence of at least one conflict. There is also an additional set of variables and clauses for encoding the subproblem of checking that S, when Q is not used, does not generate any conflict.

So, the SAT formula contains two different parts. A first part is devoted to checking that the selected set of literals S plus $\{Q\}$ is a conflict set (i.e. if $\Pi \cup W(\geq \alpha) \cup S \cup \{Q\} \vdash \perp$). This set of variables and clauses is similar to the previous one for finding almost valid arguments, but in this case is used for finding two arguments starting from a subset of $W \cup G$ and forcing the inclusion of $\{Q\}$. That is, the SAT clauses of this first part are:

(1) A clause that states that the literal Q must be true at the first step.

(2) A clause that states that at least one conflict variable c_L must be true.

(3) For every conflict variable c_L, a clause that states that if c_L is true then literals L and $\sim L$ must be true at the final step of the argument.

(4) The rest of clauses are the same ones described in the first part of the previous encoding, except the clauses of the item 5 that are not included, but now considering as possible literals and rules at every step the ones computed from the base set $W \cup G \cup \{Q\}$ and using only strict rules.

The process for computing the possible literals and rules that can be potentially applied in every step of the argument is the same forward reasoning process presented for the previous encoding. This same process is used for discovering the set of conflict variables c_L that need to be considered, because we can potentially force the conflict c_L if at the end of this process both L and $\sim L$ appear as reachable literals.

A second part of the SAT formula is devoted to checking that the selected set of variables and clauses S at the first step, without using Q, does not cause any conflict with the strict rules. So this second part of the formula contains a variable for any literal that appears in the logical closure of $G \cup W$ with respect to the strict rules. Actually, this second part of the formula is analogous to the second part of the formula for the previous encoding.

Observe that this encoding for searching conflicts for Q not only allows to check the existence of conflicts, but it also gives an explicit conflict set: the variables set to true that represent the chosen set S, together with almost valid arguments for those literals in S that have arguments in *ND*. So, we can explain the reasons for each conflict detected.

4.3. ASP Encodings for Finding Warranted Literals

Due to the recursive characterization of warranted conclusions and that they can be naturally defined as sets computed with propositional rules and certain constraints, we present a natural implementation based on a transformation to answer set programming (ASP) [Bar03]. As it has been pointed in a recent survey about the use of ASP in argumentation frameworks [TS11], to date most of the work has been centered around using ASP in abstract argumentation frameworks, although there is some theoretical work about transformations from DeLP to ASP [TKI08], but not so much in more concrete argumentation frameworks that are more suitable for applications. Our ASP transformation for the efficient resolution of queries in our argumentation framework go in the direction pointed in that survey. Moreover, with our transformation we can not only discover warranted conclusions, but also give full explanations (arguments) for the warranted conclusions, and in case of conflicts we can also explain the reasons of the conflict. Current results show that modern ASP solvers, like the one we use here: the Potassco suite [GKK$^+$11b], are extremely competitive with state-of-the-art CSP solvers on many problems [DW11b].

4.3.1. Looking for Almost Valid Arguments

In order to determine whether a given literal P has an almost valid argument which does not depend on a valid argument $\langle A, Q\rangle$, we follow some rule schemes for translating constraint satisfaction problems (CSP) to ASP [DW11b]. In particular, since our problem is to find a minimal set of arguments $M \subseteq \mathit{VA}$, with $\langle A, Q\rangle \notin M$, that satisfy Conditions (AV1) to (AV6) for P w.r.t. W and B, we propose to use a straightforward ASP encoding in which an atom of the form $e(L, i)$ is defined for each literal L with a valid argument in VA expressing the instantiation of L in M. Then, as for each literal L with a valid argument in VA the condition of belonging to the set M is Boolean, possible instantiations for the atom are encoded by the following choice rule:

$$\{e(L, 0), e(L, 1)\} \leftarrow .$$

Furthermore, we specify that L has to take at least one value:

$$\perp \leftarrow \mathit{not}\ e(L, 0), \mathit{not}\ e(L, 1)$$

and that it has to take at most one value:

$$\perp \leftarrow 2\{e(L, 0), e(L, 1)\}.$$

Then, we define inductively the following sets of variables representing literals that can be involved in the almost valid argument we are looking for. We define S_1 as a set with a variable for each valid argument which can be a subargument of an almost valid argument for P:

$$S_1 = \{v^1_L \mid \langle B, L\rangle \in \mathit{VA}\backslash\{\langle A, Q\rangle\}\}.$$

Where lately, variables in S_1 will be encoded using atoms of the form $e(v^1_L, i)$, with $i \in \{1, 0\}$. Having $e(v^1_L, 1)$ in the final answer set will mean that the argument for literal L is a subargument of the almost valid argument for P, otherwise having $e(v^1_L, 0)$ means it is not a subargument. We define another set S_2, formed by the rest of the literals of the language. They are encoded as variables $\mathit{var}(v^2_L)$. In this case we know that v^2_L is part of the argument, if it appears in the final answer set. Another set of variables representing conflicts between two literals is created. This new set, denoted as C, contains a variable for each literal L being part of sets S_1 or S_2 and having $\sim L$ in S_1 or S_2.

Having all the variables defined, program rules are defined by the following encodings:

1) For strict clauses, we encode that each appearing literal L is substituted for its corresponding predicate, depending if it is in S_1 or S_2. Having $e(v^1_L, 1)$ for the first and $\mathit{var}(v^2_L)$ for the second one.

2) Defeasible clauses are encoded in the same way strict clauses were, but a choice variable is added to the body of the rule. So, whenever the body of a defeasible clause is all True, the inclusion in the final set of the head can be avoided by setting this choice atom to False.

3)We encode a rule stating that if two literals generating a conflict in C are in the answer set, the variable representing the conflict will be as well.

4) As no conflicts can arise, integrity constraints stating that no variable in C can be in the answer set are added to the encoding.

5) An integrity constraint is added to the encoding for each blocked literal having a variable in S_1 and S_2, denoting that it can not be in the answer set.

6) In a similar way, for each warranted literal a rule stating that it must be in the answer set is added to the encoding.

7) Finally, an integrity constraint with the form $\bot \leftarrow$ *not* P, is added to the encoding to denote that P must be in the answer set.

Next we show that as long as there exists an answer set for the previous encoding schemes, the almost valid argument found during the search process will satisfy Conditions (AV1) to (AV6): [2]

(AV1) : Given that the only possible rules that can be selected for building the almost valid argument are those defined as α-rules, the conclusion of an α-rule can only become valid with strength at most α. So, the only possible subarguments with strength greater than α are the ones corresponding to literals warranted at higher levels. Observe that rule (6) in the encoding forces all such warranted literals to be in the answer set, so in the final argument for P any such subargument is implicitly used.

(AV2) : This condition is satisfied before solving the encoding with the solver. Using a polynomial process it is checked if P is in the warranted or blocked set of literals.

(AV3) : First part of this condition is assured by checking if $\sim P$ is in the set of blocked literals. The second part is assured by rules (3) and (4) in the encoding. If a literal in the initial set has its negation in $W(> \alpha)$, a conflict arises when the literal is selected. Then, if there exists a contradiction, a consistency constraint is violated.

(AV4) : Before the execution of the query we explicitly check if the literal is in the set of valid literals VA.

(AV5): For a defeasible clause to be part of an argument for P, we need all literals in the body of the rule to be True. All warranted literals

[2]Given two sets W and B of warranted and blocked literals respectively, we say that a program rule R is an α-rule if i) $N(R) > \alpha$ and $Body(R) \setminus W(> \alpha) \neq \emptyset$ or $N(R) = \alpha$, ii) $Body(R) \cap B = \emptyset$, and iii) $Head(R), \sim Head(R) \notin W \cup B$.

are forced to be True in the encoding and the rest of possible literals to be evaluated as True are those in the set S_1 which are valid literals. The last part of the condition is assured by the selection of α-rules, as no rule with the head in blocked set can be used to build an almost valid argument for P.

(AV6) : The initial set S_1 is formed by valid literals which can be activated or not. Any defeasible clause with blocked literals in the body is used as α-rule, additionally any defeasible rule with all the body with warranted literals could exist because that would mean that P is valid. Then, if an argument for P is found, it is because some literals in S_1 are activated.

4.3.2. Looking for Collective Conflicts

We present here an ASP encoding that generates a logic program that has a model if and only if the answer to the query `Conflict` is `true`, i.e. if there exists a set $G \subseteq \{P \mid \langle E, P\rangle \in VA\backslash\{\langle A, Q\rangle\} \cup ND\}$ such that $\Pi \cup W \cup G \not\vdash \bot$ and $\Pi \cup W \cup G \cup \{Q\} \vdash \bot$.

To achieve this objective, we propose to use an encoding schema structured in two parts. The first part is devoted to find a subset of valid literals which activated (True instantiation) does not generate a conflict w.r.t W. Then, in the second part, we check the condition that when the literal Q is added to the subset of valid literals, a conflict appears. Finally, we link the two parts of the encoding to assure that when one variable from the second part is activated, the analogous variable for the same literal in the first part is activated as well.

Next we formalize the encoding for the first part. We define the sets of variables representing literals involved in the process of checking whether the strict clauses without the need of Q, generates a conflict. The starting set of literals is $G_1 = W \cup \{P \mid \langle E, P\rangle \in VA\backslash\{\langle A, Q\rangle\} \cup ND\}$. So we define the set $S^{e,1}$ in the following way:

$$S^{e,1} = \{v_L^{e,1} | L \in G_1\}$$

Each variable in $S^{e,1}$ is encoded as an atom with the form $e(v_L^{e,1}, i)$, having $i \in \{1, 0\}$. As done previously, a set of variables $S^{v,1}$ encodes all those literals appearing in the rules and not in G_1. We also define a set of conflicts C_1 in the same way that we did in the almost valid ASP encoding.

Variable and program rules for this first part of the encoding are defined as follows:

1) We encode strict clauses replacing literals by variables defined in $S^{e,1}$ and $S^{v,1}$.

2) For each conflict in C_1, we encode a rule denoting that if both two opposite literals are in the answer set, the conflict variable will be as well.

3) Integrity constraints stating that no conflict variable can be in the answer set.

4) Rules stating that warranted literals must be in the answer set.

Following a similar encoding for the second part, we define analogous sets for checking whether a conflict can be generated with the strict clauses but now using the literal Q. So, now the starting set of literals is $G_2 = G_1 \cup \{\langle A, Q\rangle\}$. This is done to distinguish, between literals activated by strict clauses, and literals activated during the search of the conflict. Then, we define the sets $S^{v,2}$ and $S^{e,2}$ in the following way:

$$\begin{array}{l} S^{v,2} = \{v_L^{v,2} \mid L \in G_2 \cup \{\mathit{Head}(R), \mathit{Body}(R) \mid R \in \Pi\}\} \\ S^{e,2} = \{v_L^{e,2} \mid L \in G_2\} \end{array}$$

We also create the set of variables C_2 representing conflicts in the set $S^{v,2}$. An special variable CC is defined to denote that a collective conflict exists.

Variable and program constraints for this second part of the encoding are defined as follows:

1) We encode strict clauses replacing literals by variables defined in $S^{v,2}$ and $S^{e,2}$. Being $var(v_L^{v,2})$ and $e(v_L^{e,2}, 1)$ the proper translations, respectively.

2) For each conflict in C_2, we encode a rule denoting that if both two opposite literals are in the answer set, the conflict variable will be as well.

3) A cardinality constraint stating that if at least a conflict in C_2 is in the answer set, the variable CC will be as well.

4) An integrity constraint stating: $\bot \leftarrow \mathit{not}\ CC$.

5) Rules stating that warranted literals must be in the answer set.

Finally, we define the following linking rules between the variable sets of the two parts of the encoding:

1) For each variable representing a literal in the set $S^{e,2}$, a rule stating that if this variable is in the answer set, a variable representing the same literal must be in $S^{e,1}$.

2) For each variable representing a literal in the set $S^{e,2}$, a rule stating that if this variable is in the answer set, a variable representing the same literal must be in $S^{v,2}$.

5 The Maximal Ideal Output for RP-DeLP Framework

In the previous chapter we showed an algorithm for computing the set of conclusions that can be ultimately warranted in RP-DeLP programs with multiple outputs.

Now in this chapter we are interested in characterize an unique output property for the particular framework of RP-DeLP programs. The usual skeptical approach would be to adopt he intersection of all possible outputs. However, in addition to the computational limitation, as stated in [Pol09], adopting the intersection of all outputs may lead to an inconsistent output (in the sense of violating the base of the underlying recursive warrant semantics) in case some particular recursive situation among literals of a program occurs. Intuitively, for a conclusion, to be in the intersection does not guarantee the existence of an argument for it that is recursively based on ultimately warranted conclusions.

Contents

5.1. Characterization of the Maximal Ideal Output for an RP-DeLP Program

For instance, consider the following situation involving three conclusions P, Q, and T, where P can be warranted whenever Q is blocked, and vice-versa. Moreover, suppose that T can be warranted when either P or Q are warranted. Then, according to the warrant recursive semantics, we would get two different outputs: one where P and T are warranted and Q is blocked, and the other one where Q and T are warranted and P is blocked. Then, adopting the intersection of both outputs we would get that T would be ultimately warranted, however T should be in fact rejected since neither P nor Q are ultimately warranted conclusions.

According to this example, one could take then as the set of ultimately warranted conclusions of RP-DeLP programs those conclusions in the intersection of all outputs which are recursively based on ultimately warranted conclusions. However, as in RP-DeLP there might be different levels of defeasibility, this approach could lead to an incomplete solution, in the sense of not being the biggest set of ultimately warranted conclusions with maximum strength.

For instance consider the above example extended with two defeasibility levels as follows. Suppose that P can be warranted with strength α whenever Q is blocked, and vice-versa. Moreover, suppose that T can be warranted with strength α whenever P is warranted at least with strength α, and that T can be warranted with strength β, with $\beta < \alpha$, independently of the status of conclusions P and Q. Then, again we get two different outputs: one output warrants conclusions P and T with strength α and blocks conclusion Q, and the other one warrants conclusions Q and T with strengths α and β, respectively, and blocks P. Now, by adopting conclusions of the intersection which are recursively based on ultimately warranted conclusions, we get that conclusion T is finally rejected, since T is warranted with a different argument and strength in each output. However, as we are interested in determining the biggest set of warranted conclusions with maximum strength, it seems quite reasonable to reject T at level α but to warrant it at level β.

Therefore, we are led to define the *maximal ideal output* for an RP-DeLP program $\mathcal{P} = (\Pi, \Delta, \preceq)$ as a pair $(\mathit{Warr}, \mathit{Block})$ of respectively warranted and blocked conclusions, with a maximum strength, such that:

(i) the arguments of all conclusions in $\mathit{Warr} \cup \mathit{Block}$ are recursively based on warranted conclusions;

(ii) a conclusion is warranted (at level α) if does not generate any conflict with the set of already warranted conclusions (at a level $\beta > \alpha$) and it is not involved in any cycle of a warrant dependency graph; otherwise, it is blocked; and

(iii) a conclusion is rejected if it can be neither warranted nor blocked to any level.

In fact, in a different context, this idea corresponds to the *maximal ideal extension* defined by Dung, Mancarella and Toni [DMT06, DMT07] as an alternative skeptical basis for defining collections of justified arguments in the abstract argumentation frameworks promoted by Dung [Dun95b] and Bondarenko *et al.* [BDKT97].

Definition 5.1 (Maximal ideal output)**.** *The* maximal ideal output *for an RP-DeLP program* $\mathcal{P} = (\Pi, \Delta, \preceq)$*, with defeasibility levels* $1 > \alpha_1 > \ldots > \alpha_p$*, is a pair* $(\mathit{Warr}, \mathit{Block})$ *such that, for every valid argument* $\langle A, Q \rangle$ *of strength* α_i *with respect to* $\mathit{Warr}(\geq \alpha_i)$ *and* $\mathit{Block}(> \alpha_i)$*, the following recursive constraint is satisfied:*

1. $Q \in \mathit{Block}(\alpha_i)$ *whenever one of the two following cases holds:*

   <u>Case 1</u> There exists a set $\mathbb{G}$ *of valid arguments of strength* α_i *such that the two following conditions hold:*

   (i) $\langle A, Q \rangle \not\sqsubset \mathbb{G}$*, and*

   (ii) $\mathbb{G} \cup \{\langle A, Q \rangle\}$ *minimally conflicts with respect to the set* $W = \mathit{Warr}(> \alpha_i) \cup \{P \mid \langle B, P \rangle \sqsubset \mathbb{G} \cup \{\langle A, Q \rangle\}\}$.

   <u>Case 2</u> There exists a set $\mathbb{H}$ *of valid arguments of strength* α_i *such that the three following conditions hold:*

   (i) $\langle A, Q \rangle \not\sqsubset \mathbb{H}$.

   (ii) There exists a set of arguments $\mathbb{F}$ *of strength* α_i *that are almost valid with respect to* $\mathbb{H} \cup \langle A, Q \rangle$ *and such that there is a cycle in the warrant dependence graph* (V, E) *for* $\mathbb{H} \cup \langle A, Q \rangle$ *and* $\mathbb{F}$*, and any argument* $\langle C, R \rangle \in \mathbb{H}$ *is such that* R *is either a vertex of the cycle or* $\langle C, R \rangle$ *does not satisfy Case 1.*

   (iii) For some vertex $v \in V$ *of the cycle, either* v *is the vertex of conclusion* Q *or* v *is the vertex of some other conclusion in* $\mathbb{H}$*, and there exists a path from* v *to the the vertex of conclusion* Q*.*

2. *Otherwise,* $Q \in \mathit{Warr}(\alpha_i)$*.*

The intuition underlying the maximal ideal output definition is as follows. The conclusion of every valid (not rejected) argument $\langle A, Q \rangle$ of strength α_i is either warranted or blocked. Then, it is eventually blocked if either (Case 1) it is involved in some conflict with respect to $\mathit{Warr}(> \alpha_i)$ and a set $\mathbb{G}$ of valid arguments whose supports do not depend on $\langle A, Q\rangle$, or (Case 2) the warranty of $\langle A, Q\rangle$ depends on some circular definition of conflict between a set of valid arguments $\mathbb{H}$ whose supports do not depend on $\langle A, Q\rangle$ and a set of almost valid arguments $\mathbb{F}$ whose supports depend on some argument in $\mathbb{H} \cup \langle A, Q\rangle$. In fact, the idea here is that if the warranty of $\langle A, Q\rangle$ depends on some circular definition of conflict between the arguments of $\mathbb{H}$ and $\mathbb{F}$, one could consider two different outputs (status) for conclusion Q: one with Q warranted and another one with Q blocked. Therefore, conclusion Q is blocked for the maximal ideal output. In general, the arguments of $\mathbb{H}$ and $\mathbb{F}$ involved in a cycle are respectively warranted and rejected for the maximal ideal output.

For instance, consider again the recursion case in the Example 4 from Section 3.2. Figure 3.2 shows the warrant dependency graph for the set of valid arguments

$$\mathbb{H} = \{\langle\{p\}, p\rangle, \langle\{q\}, q\rangle, \langle\{r\}, r\rangle\}$$

and the set of almost valid arguments

$$\mathbb{F} = \{\langle\{q, r, q \wedge r \to \sim p\}, \sim p\rangle, \langle\{p, r, p \wedge r \to \sim q\}, \sim q\rangle, \langle\{p, q, p \wedge q \to \sim r\}, \sim r\rangle\}$$

Then, since for every valid argument in $\mathbb{H}$ there is a a cycle, the maximal ideal output for the RP-DeLP program is $\mathit{Warr} = \emptyset$ and $\mathit{Block} = \{p, q, r\}$.

As a matter of another example, we have the following example:

Example 7. *Consider the RP-DeLP program* $\mathcal{P}_7 = (\Pi, \Delta, \preceq)$ *with*

$$\Pi = \{y, \sim y \leftarrow p \wedge r, \sim y \leftarrow q \wedge s\} \text{ and } \Delta = \{p, q, r \leftarrow q, s \leftarrow p\},$$

and a single defeasibility level α *for* Δ.
Consider the sets $W(1) = \{Q \mid \Pi \vdash_R Q\} = \{y\}$, $B(1) = \emptyset$, $W(\alpha) = \emptyset$ *and* $B(\alpha) = \emptyset$. *Now consider arguments for conclusions* p *and* q*; i.e.*

$$\mathcal{H}_1 = \langle\{p\}, p\rangle \text{ and } \mathcal{H}_2 = \langle\{q\}, q\rangle.$$

Finally, consider arguments for conclusions r *and* s*; i.e.*

$$\mathcal{F}_1 = \langle\{q, r \leftarrow q\}, r\rangle \text{ and } \mathcal{F}_2 = \langle\{p, s \leftarrow p\}, s\rangle.$$

Obviously, $\mathcal{H}_1$ *and* $\mathcal{H}_2$ *are valid arguments with respect to* $W(\geq \alpha)$ *and* $B(> \alpha)$*, and* $\mathcal{F}_1$ *and* $\mathcal{F}_2$ *are almost valid arguments with respect to* $\{\mathcal{H}_1, \mathcal{H}_2\}$. *Figure 5.1 shows the warrant dependency graph for* $\{\mathcal{H}_1, \mathcal{H}_2\}$ *and* $\{\mathcal{F}_1, \mathcal{F}_2\}$. *The cycle of the graph expresses that (1) the warranty of* p *depends on a (possible) conflict with* r*; (2) the support of* r *depends on* q*; (3) the warranty of* q *depends on a (possible) conflict with* s*; and (4) the support of* s *depends on* p.

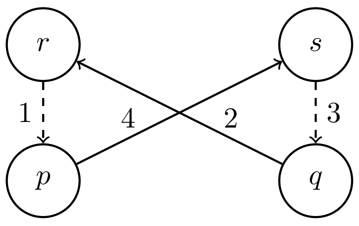


Figure 5.1: Warrant dependency graph for RP-DeLP program $\mathcal{P}_7$ from example 7.

Figure 5.1 shows the warrant dependency graph for the set of valid arguments

$$\mathbb{H} = \{\langle\{p\}, p\rangle, \langle\{q\}, q\rangle\}$$

and the set of almost valid arguments

$$\mathbb{F} = \{\langle\{q, r \leftarrow q\}, r\rangle, \langle\{p, s \leftarrow p\}, s\rangle\}.$$

Again, since for every valid argument there is a cycle, the maximal ideal output for $\mathcal{P}_7$, is $Warr = \{y\}$ and $Block = \{p, q\}$.

Now consider a new program $\mathcal{P}_8 = (\Pi, \Delta, \preceq)$, which is an extension of $\mathcal{P}_7$ with the following set of defeasible rules:

$$\Pi = \{y, {\sim}y \leftarrow p \wedge r, {\sim}y \leftarrow q \wedge s\},\ \Delta = \{p, q, t, r \leftarrow q, s \leftarrow p, t \leftarrow p, t \leftarrow q\},$$

and with Δ being stratified as follows:

$$\text{level } \alpha_1\text{: } \{p, q, r \leftarrow q, s \leftarrow p, t \leftarrow p, t \leftarrow q\} \qquad \text{level } \alpha_2\text{: } \{t\}.$$

Assume α_1 is the corresponding highest level and α_2 is the lowest level, with $1 > \alpha_1 > \alpha_2 > 0$. Obviously, $Warr(1) = \{y\}$ and, at level α_1, $\mathcal{H}_1 = \langle\{p\}, p\rangle$ and $\mathcal{H}_2 = \langle\{q\}, q\rangle$ are valid arguments. Moreover, $\mathcal{F}_1 = \langle\{q, r \leftarrow q\}, r\rangle$, $\mathcal{F}_2 = \langle\{p, s \leftarrow p\}, s\rangle$, $\mathcal{F}_3 = \langle\{q, t \leftarrow q\}, t\rangle$ and $\mathcal{F}_4 = \langle\{p, t \leftarrow p\}, t\rangle$ are almost valid arguments with respect to $\{\mathcal{H}_1, \mathcal{H}_2\}$. Figure 5.2 shows the warrant dependency graph for $\{\mathcal{H}_1, \mathcal{H}_2\}$ and $\{\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3, \mathcal{F}_4\}$. As for every valid argument there is a cycle, p and q are blocked, while r and s are rejected for the maximal ideal output since the support of $\mathcal{F}_1$ depends on q and the support of $\mathcal{F}_2$ depends on p, and p and q are blocked. Remark that t is also rejected at level α_1 since the support of $\mathcal{F}_3$ depends on p, the support of $\mathcal{F}_4$ depends on q, and p and q are blocked. Therefore, $Warr(1) = \{y\}$, $Warr(\alpha_1) = \emptyset$ and $Block(\alpha_1) = \{p, q\}$. Finally, at level α_2, $\langle\{t\}, t\rangle$ is the unique valid argument and therefore t is warranted, hence, $Warr(\alpha_2) = \{t\}$ and $Block(\alpha_2) = \emptyset$. Therefore, the maximal ideal output for $\mathcal{P}_8$ is $Warr = \{y, t\}$ and $Block = \{p, q\}$.

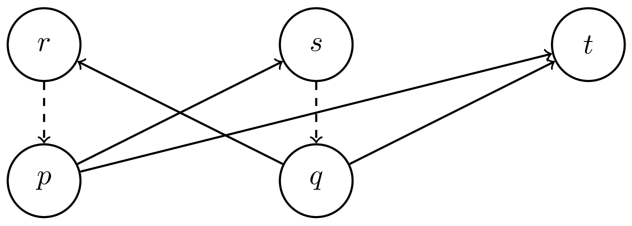


Figure 5.2: Warrant dependency graph for RP-DeLP program $\mathcal{P}_8$ from example 7

.

Next proposition shows that the maximal ideal output for an RP-DeLP program is unique.

Proposition 5.1 (Unicity of the maximal ideal output)**.** *Let $\mathcal{P} = (\Pi, \Delta, \preceq)$ be an RP-DeLP program. The pair $(\mathit{Warr}, \mathit{Block})$ of warranted and blocked conclusions that satisfies the maximal ideal output characterization for $\mathcal{P}$ of Definition 5.1 is unique.*

Proof. Suppose that $(\mathit{Warr}, \mathit{Block})$ and $(\mathit{Warr}', \mathit{Block}')$ are pairs of warranted and blocked conclusions that satisfy the maximal ideal output characterization for $\mathcal{P}$ stated in Def. 5.1. Obviously, $\mathit{Warr}(1) = \mathit{Warr}'(1)$. Suppose that for some α, $\mathit{Warr}(\alpha) \neq \mathit{Warr}'(\alpha)$ and $\mathit{Warr}(\beta) = \mathit{Warr}'(\beta)$, for all $\beta > \alpha$. As $\mathit{Warr}(\alpha) \neq \mathit{Warr}'(\alpha)$, suppose that $\langle A, Q\rangle$ of strength α is valid with respect to $(\mathit{Warr}, \mathit{Block})$ and $(\mathit{Warr}', \mathit{Block}')$ but $Q \notin \mathit{Warr}(\alpha)$ and $Q \in \mathit{Warr}'(\alpha)$. Then, $Q \in \mathit{Block}(\alpha)$ and $\langle A, Q\rangle$ is either

Case 1: involved in a conflict with respect to $\mathit{Warr}(> \alpha)$ and a set $\mathbb{G}$ of valid arguments of strength α which supports do not depend on $\langle A, Q\rangle$, or

Case 2: the warranty of $\langle A, Q\rangle$ depends on a circular definition of conflict between a set $\mathbb{H}$ of valid arguments which supports do not depend on $\langle A, Q\rangle$ and a set $\mathbb{F}$ of almost valid arguments which supports depend on some argument in $\mathbb{H} \cup \langle A, Q\rangle$.

Moreover, as $Q \in \mathit{Warr}'(\alpha)$, $\langle A, Q\rangle$ is not involved in a conflict nor in a cycle with respect to $\mathit{Warr}'(> \alpha)$.

As all sets $\mathbb{G}$ and $\mathbb{H}$ of valid arguments of strength α whose supports do not depend on $\langle A, Q\rangle$ are also valid with respect to $(\mathit{Warr}', \mathit{Block}')$, and all sets $\mathbb{G}'$ and $\mathbb{H}'$ of valid arguments of strength α which supports do not depend on $\langle A, Q\rangle$ are also valid with respect to $(\mathit{Warr}, \mathit{Block})$, there should exist at least an argument $\langle B, P\rangle$ such that

(i) it is almost valid with respect to a set $\mathbb{H}$ of valid arguments that satisfy Condition (b) for argument $\langle A, Q\rangle$ and output $(\mathit{Warr}, \mathit{Block})$, and

(ii) it is not almost valid with respect to $\mathbb{H}$.

Therefore, $\langle B, P\rangle$ should violate Condition (AV5) from Section 3.2 with respect to $\mathbb{H}$ and $Warr'$ and $Block'$, and thus, for some subargument $\langle C, R\rangle \sqsubset \langle B, P\rangle$ of strength α it must hold that $R \not\in Warr'(\alpha)$ and $\langle C, R\rangle \not\in \mathbb{H}$ and R or $\sim R \in Block'(\geq \alpha)$. Now, as $\langle C, R\rangle \not\in \mathbb{H}$ and $\langle B, P\rangle$ is almost valid with respect to $\mathbb{H}$, either $R \in Warr(\alpha)$, or $R, \sim R \not\in Block(\geq \alpha)$. If $R \in Warr(\alpha)$, because of the recursive warrant semantics, $\langle A, Q\rangle \not\sqsubset \langle C, R\rangle$, and thus, $R \in Warr'(\alpha)$. If $R \not\in Warr(\alpha)$, we have $R, \sim R \not\in Block(\geq \alpha)$ and R or $\sim R \in Block'(\geq \alpha)$. As $Block(\beta) = Block'(\beta)$ for all $\beta > \alpha$, $R, \sim R \not\in Block(\alpha)$ and R or $\sim R \in Block'(\alpha)$. Then either $R \in Warr(\alpha)$ or $\langle C, R\rangle$ is not valid with respect to $(Warr, Block)$, and thus, $\langle A, Q\rangle \sqsubset \langle C, R\rangle$. Now, as the warranty of $\langle A, Q\rangle$ depends on a circular definition of conflict between the set $\mathbb{H}$ and a set $\mathbb{F}$ of almost valid arguments which supports depend on some argument in $\mathbb{H} \cup \langle A, Q\rangle$ with $\langle B, P\rangle \in \mathbb{F}$, there is a cycle in the warrant dependence graph (V, E) for $\mathbb{H}$ and $\mathbb{F}$ and any argument $\mathcal{C} \in \mathbb{H}$ is such that the conclusion of $\mathcal{C}$ is either a vertex of the cycle or $\mathcal{C}$ does not satisfy Condition (a). Then, if R or $\sim R \in Block'(\alpha)$ and $\langle C, R\rangle \not\in \mathbb{H}$, R or $\sim R \in Block(\alpha)$. Hence, $Warr(\alpha) = Warr'(\alpha)$ and $Block(\alpha) = Block'(\alpha)$ for all defeasibility level α. ∎

When we restrict ourselves to the case of RP-DeLP programs with a single defeasibility level, we get the following property of the maximal ideal output.

Proposition 5.2 (Programs with a single defeasibility level)**.** *Let $\mathcal{P}$ be an RP-DeLP program with a single defeasibility level, and let $(Warr, Block)$ be the maximal ideal output for $\mathcal{P}$. Then, for each output $(Warr', Block')$ for $\mathcal{P}$, we have $Warr \subseteq Warr'$ and $Block \subseteq Warr' \cup Block'$.*

Proof. Obviously, $Warr(1) = Warr'(1)$, for each output $(Warr', Block')$ for $\mathcal{P}$. Since we are considering a single defeasibility level, $\langle A, Q\rangle$ is a valid argument with respect to $Warr$ iff $P \in Warr$ for all $\langle B, P\rangle \sqsubset \langle A, Q\rangle$. Suppose that $\langle A, \varphi\rangle$ is valid with respect to $Warr$ and not valid with respect to $Warr'$. Then, there should exist an argument $\langle B, P\rangle$ such that $\langle B, P\rangle \sqsubset \langle A, Q\rangle$ and $\langle B, P\rangle \in Warr$ but, $\langle B, P\rangle \not\in Warr'$ and $\langle B, P\rangle$ is valid with respect to $Warr'$. Hence, there should exist a set of arguments $\mathbb{G}$ valid with respect to $Warr'$ such that $\langle B, P\rangle \not\sqsubset \mathbb{G}$ and $\{\langle B, P\rangle\} \cup \mathbb{G}$ minimally conflicts with respect to the set $W = \{R \mid \langle C, R\rangle \sqsubset \mathbb{G} \cup \{\langle B, P\rangle\}\}$. If each argument in $\mathbb{G}$ was valid with respect to $Warr$, then $\{\langle B, P\rangle\} \not\in Warr$. Then, there should exist an argument $\langle C, R\rangle \in \mathbb{G}$ such that $\langle C, R\rangle$ is valid with respect to $Warr'$ and not valid with respect to $Warr$, and thus, there should exist an argument $\langle D, T\rangle$ such that $\langle D, T\rangle \sqsubset \langle C, R\rangle$ and $\langle D, T\rangle \in Warr'$ but, $\langle D, T\rangle \not\in Warr$ and $\langle B, P\rangle$ is valid with respect to $Warr$. Hence, there should exist a cycle in a warrant dependency graph and vertices for $\langle B, P\rangle$

and $\langle C, R\rangle$ should be vertices of the cycle and there should exist a path from some vertex of the cycle to the vertex of $\langle B, P\rangle$, and thus, $\langle B, P\rangle \not\in Warr$. Hence, $Warr \subseteq Warr'$. Finally, as each argument $\langle A, Q\rangle$ valid with respect to $Warr$ is also valid with respect to $Warr'$ and each valid argument is either warranted or blocked, $Block \subseteq Warr' \cup Block'$. ■

The following example shows that in case we consider multiple defeasibility levels for Δ, a conclusion can be warranted for the maximal ideal output at some level α and, due to the set of warranted conclusions at higher levels, rejected for each output (extension).

Example 8. *Consider the RP-DeLP program $\mathcal{P}_9 = (\Pi, \Delta, \preceq)$ with*

$$\Pi = \{y, {\sim}y \leftarrow p \wedge s, {\sim}y \leftarrow q \wedge s\} \textit{ and } \Delta = \{p, q, {\sim}q \leftarrow p, {\sim}p \leftarrow q, s\},$$

where Δ is stratified in two defeasibility levels ($1 > \alpha_1 > \alpha_2 > 0$) as follows:

$$\textit{level } \alpha_1\textit{: } \{p, q, {\sim}q \leftarrow p, {\sim}p \leftarrow q\} \qquad \textit{level } \alpha_2\textit{: } \{s\}.$$

Obviously, $Warr(1) = \{y\}$. Then, at level α_1, we have two valid arguments:

$$\mathcal{H}_1 = \langle\{p\}, p\rangle \textit{ and } \mathcal{H}_2 = \langle\{q\}, q\rangle.$$

and two almost valid arguments with respect to $\{\mathcal{H}_1, \mathcal{H}_2\}$:

$$\mathcal{F}_1 = \langle\{p, {\sim}q \leftarrow p\}, {\sim}q\rangle \textit{ and } \mathcal{F}_2 = \langle\{q, {\sim}p \leftarrow q\}, {\sim}p\rangle.$$

Figure 5.3 shows the warrant dependency graph for $\{\mathcal{H}_1, \mathcal{H}_2\}$ and $\{\mathcal{F}_1, \mathcal{F}_2\}$. The cycle expresses that either p or q can be warranted, but not both. Hence, at level α_1, we have two possible outputs for $\mathcal{P}_9$:

$$\begin{array}{ll} Warr_1(\alpha_1) = \{p\}, & Block_1(\alpha_1) = \{q, {\sim}q\}, \\ Warr_2(\alpha_1) = \{q\}, & Block_2(\alpha_1) = \{p, {\sim}p\}. \end{array}$$

Then at level α_2, the argument $\langle\{s\}, s\rangle$ violates Condition (V3) from Section 3.2, and thus, s is rejected in both outputs. Therefore, the two possible outputs for $\mathcal{P}_9$ are:

$$\begin{array}{ll} Warr_1 = \{y, p\}, & Block_1 = \{q, {\sim}q\}, \\ Warr_2 = \{y, q\}, & Block_2 = \{p, {\sim}p\}. \end{array}$$

Let us consider now the maximal ideal output for $\mathcal{P}_9$, in which valid arguments involved in cycles are blocked and almost valid arguments involved in cycles are rejected. Obviously $Warr_{maximal}(1) = \{y\}$, and at level α_1 the maximal ideal output for $\mathcal{P}_9$ is:

$$Warr_{maximal}(\alpha_1) = \emptyset, \qquad Block_{maximal}(\alpha_1) = \{p, q\}.$$

Now, at level α_2 we have that the argument $\langle\{s\}, s\rangle$ is valid and it is not involved in a cycle nor in a conflict, and thus, s is warranted at level α_2 for the maximal ideal output. Hence, the maximal ideal output for $\mathcal{P}_9$ is:

$$Warr_{maximal} = \{y, s\}, \qquad Block_{maximal} = \{p, q\}.$$

Therefore, in the program $\mathcal{P}_9$, s is not warranted in any of their two outputs $Warr_1$ and $Warr_2$, but still s is warranted in the maximal ideal output.

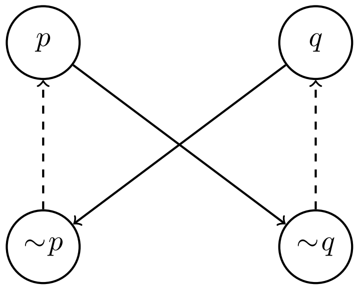


Figure 5.3: Warrant dependency graph for RP-DeLP program from example 8.

Next we will present two examples to show how warranted and blocked conclusions at higher levels are taken into account in lower levels.

Example 9. *Consider the RP-DeLP program $\mathcal{P}_{10} = (\Pi, \Delta, \preceq)$ with*

$$\Pi = \{y, \sim y \leftarrow p \wedge r, \sim y \leftarrow q \wedge s\} \text{ and } \Delta = \{p, q, r \leftarrow q, s \leftarrow p, u \leftarrow q\},$$

where Δ is stratified in two defeasibility levels ($1 > \alpha_1 > \alpha_2 > 0$) as follows:

$$\text{level } \alpha_1\text{: } \{p, q, r \leftarrow q, s \leftarrow p\} \qquad \text{level } \alpha_2\text{: } \{u \leftarrow q\}.$$

Note that $\mathcal{P}_{10}$ and $\mathcal{P}_7$ have the same strict knowledge and the same defeasible knowledge at level α_1 , the only difference is at level α_2.

So, like in $\mathcal{P}_7$, $\mathcal{P}_{10}$ has $Warr(1) = \{y\}$ and, at level α_1, $\mathcal{H}_1 = \langle\{p\}, p\rangle$ and $\mathcal{H}_2 = \langle\{q\}, q\rangle$ are valid arguments.

Moreover, $\mathcal{F}_1 = \langle\{q, r \leftarrow q\}, r\rangle$ and $\mathcal{F}_2 = \langle\{p, s \leftarrow p\}, s\rangle$ are almost valid arguments with respect to $\{\mathcal{H}_1, \mathcal{H}_2\}$.

As for every valid argument there is a cycle, p and q are blocked, while r and s are rejected for the maximal ideal output since the support of $\mathcal{F}_1$ depends on q and the support of $\mathcal{F}_2$ depends on p, and p and q are blocked. Therefore, $Warr(1) = \{y\}$, $Warr(\alpha_1) = \emptyset$ and $Block(\alpha_1) = \{p, q\}$.

However, at level α_2, argument $\mathcal{H}_3 = \langle\{q, u \leftarrow q\}, u\rangle$ is not a valid argument because it violates condition $V3$ as $q \in Block(> \alpha_2)$. Therefore the maximal ideal output for $\mathcal{P}_{10}$ is $Warr = \{y\}$ and $Block = \{p, q\}$.

Example 10. *Consider the RP-DeLP program* $\mathcal{P}_{11} = (\Pi, \Delta, \preceq)$ *with*

$$\Pi = \{y, \sim y \leftarrow p \wedge r, \sim y \leftarrow q \wedge s\} \text{ and } \Delta = \{p, q, u, r \leftarrow q, s \leftarrow p, \sim q \leftarrow u\},$$

where Δ *is stratified in two defeasibility levels* $(1 > \alpha_1 > \alpha_2 > 0)$ *as follows:*

$$\text{level } \alpha_1\text{: } \{p, q, r \leftarrow q, s \leftarrow p\} \qquad \text{level } \alpha_2\text{: } \{u, \sim q \leftarrow u\}.$$

As in the previous example $\mathcal{P}_{10}$*, at level* α_1 *we will have:* $\mathit{Warr}(1) = \{y\}$, $\mathit{Warr}(\alpha_1) = \emptyset$ *and* $\mathit{Block}(\alpha_1) = \{p, q\}$.

At level α_2 *we will have a different situation, because* $\mathit{Warr}(\alpha_2) = \{u\}$ *but* $\mathcal{H}_1 = \langle\{u, \sim q \leftarrow u\}, \sim q\rangle$ *is not a valid argument because* $q \in \mathit{Block}(\alpha_1)$ *and then, it violates condition* $V3$*. Therefore the maximal ideal output for* $\mathcal{P}_{11}$ *is* $\mathit{Warr} = \{y, u\}$ *and* $\mathit{Block} = \{p, q\}$.

Previous examples show that in case we consider multiple defeasibility levels, the set of conclusions that are warranted and blocked at each level is decisive for determining which arguments are valid at lower levels. Then, since the maximal ideal output for an RP-DeLP program corresponds to a skeptical criterion regarding warranted conclusions, it is very interesting to analyze the status of the Closure Postulate for the maximal ideal output for RP-DeLP programs with multiple defeasibility levels.

Proposition 5.3 (Closure for the maximal ideal output). *Let* $\mathcal{P} = (\Pi, \Delta, \preceq)$ *be an RP-DeLP program with defeasibility levels* $1 > \alpha_1 > \ldots > \alpha_p > 0$*, and let* $(\mathit{Warr}, \mathit{Block})$ *be the maximal ideal output for* $\mathcal{P}$*. Then, if* $\Pi \cup \mathit{Warr}(\geq \alpha_i) \vdash_R Q$ *and* $\Pi \cup \mathit{Warr}(> \alpha_i) \not\vdash_R Q$*, then either* $Q \in \mathit{Warr}(\alpha_i)$*, or* $Q \in \mathit{Block}(> \alpha_i)$*, or* $\sim Q \in \mathit{Block}(> \alpha_i)$.

Proof. Suppose that for some α_i, $\Pi \cup \mathit{Warr}(\geq \alpha_i) \vdash_R Q$, $\Pi \cup \mathit{Warr}(> \alpha_i) \not\vdash_R Q$, $Q \notin \mathit{Warr}(\alpha_i)$, and $Q, \sim Q \notin \mathit{Block}(> \alpha_i)$. Then, since $\Pi \cup \mathit{Warr} \not\vdash \bot$, $\Pi \cup \mathit{Warr}(\geq \alpha_i) \cup \{Q\} \not\vdash \bot$, there exists a valid argument $\langle A, Q\rangle$ for Q of strength α_i. Now, since $Q \notin \mathit{Warr}(\alpha_i)$, according to Def. 5.1 there are two possible cases:

<u>Case 1</u> There is a set $\mathbb{G}$ of valid arguments of strength α_i such that (i) $\langle A, Q\rangle \not\sqsubseteq \mathbb{G}$, and (ii) $\mathbb{G} \cup \{\langle A, Q\rangle\}$ generates a conflict with respect to $W = \mathit{Warr}(> \alpha_i) \cup \{P \mid \langle B, P\rangle \sqsubset \mathbb{G} \cup \{\langle A, Q\rangle\}\}$. If $\mathbb{G} \cup \{\langle A, Q\rangle\}$ generates a conflict with respect to W, Conditions (C) and (M) hold for W, and thus, $\Pi \cup W \cup \{Q\} \cup \{P \mid \langle B, P\rangle \in \mathbb{G}\} \vdash \bot$ and $\Pi \cup W \cup S \not\vdash \bot$, for all $S \subset \{Q\} \cup \{P \mid \langle B, P\rangle \in \mathbb{G}\}$. Consider $W' = \{R \mid \langle B, R\rangle \sqsubset \langle A, Q\rangle\}$. Then, as $W' \subseteq W$ and $\Pi \cup W' \vdash_R Q$, if $\Pi \cup W \cup \{Q\} \cup \{P \mid \langle B, P\rangle \in G\} \vdash \bot$, then $\Pi \cup W \cup \{P \mid \langle B, P\rangle \in \mathbb{G}\} \vdash \bot$, and thus, either Q is warranted at level α_i or Q is rejected at level α_i because Q or $\sim Q$ are blocked at a level β with $\beta > \alpha_i$. In other words, either $Q \in \mathit{Warr}(\alpha_i)$, or $Q \in \mathit{Block}(> \alpha_i)$, or $\sim Q \in \mathit{Block}(> \alpha_i)$.

Case 2 There is a set of valid arguments $\mathbb{H}$ of strength α_i such that (i) there is a set of arguments $\mathbb{F}$ of strength α_i that are almost valid with respect to $\mathbb{H} \cup \{\langle A, Q\rangle\}$, (ii) there is a cycle in the warrant dependence graph (V, E) for $\mathbb{H} \cup \{\langle A, Q\rangle\}$ and $\mathbb{F}$, and any argument $\langle C, R\rangle \in \mathbb{H}$ is either a vertex of the cycle or $\langle C, R\rangle$ does not generate any conflict, and (iii) the vertex v_Q for Q is a vertex of the cycle or there is a path from a vertex for some conclusion in $\mathbb{H}$ to v_Q. Then, according to Def. 3.1, there is an almost valid argument for conclusion $\sim Q$ in $\mathbb{F}$ or an strict rule $\sim Q \leftarrow L_1 \wedge \ldots \wedge L_m \in \Pi$ such that $\{L_1, \ldots, L_m\} \subseteq \mathit{Warr}(\geq \alpha_i) \cup \{H \mid \langle E, H\rangle \in \mathbb{H}\} \cup \{F \mid \langle J, F\rangle \in \mathbb{F}\}$, and thus, there is an almost valid argument $\langle D, \sim Q\rangle$ for conclusion $\sim Q$ in $\mathbb{F}$, and an edge from the vertex $v_{\sim Q}$ to the vertex v_Q. Now, since $\Pi \cup \mathit{Warr}(\geq \alpha_i) \vdash_R Q$ and $Q \notin \mathit{Warr}(\geq \alpha_i)$, there exists a strict rule $Q \leftarrow L'_1 \wedge \ldots \wedge L'_p \in \Pi$ with all the L'_j's in $\mathit{Warr}(\geq \alpha_i)$. Moreover, as $\Pi \cup \mathit{Warr}(> \alpha_i) \not\vdash_R Q$, there is at least one literal $L' \in \{L'_1, \ldots, L'_p\}$ such that $L' \in \mathit{Warr}(\alpha_i)$, and thus, there is a valid argument $\langle J, L'\rangle$ for L' of strength α_i and $\langle A, Q\rangle \not\sqsubset \langle J, L'\rangle$. Then, there is a cycle in the warrant dependence graph (V', E') for $\mathbb{H} \cup \{\langle A, Q\rangle\} \cup \{\langle J, L'\rangle\}$ and $\mathbb{F}$ and an edge from the vertex $v_{\sim Q}$ to the vertex $v_{L'}$, and thus, $L' \notin \mathit{Warr}(\alpha_i)$. Hence, either $Q \in \mathit{Warr}(\alpha_i)$, or $Q \in \mathit{Block}(> \alpha_i)$, or $\sim Q \in \mathit{Block}(> \alpha_i)$. ∎

As a direct consequence, we have the following simpler form of the Closure Postulate for the particular case of programs with a single defeasibility level.

Corollary 5.4 (Closure for RP-DeLP programs with a single defeasibility level)**.** *Let $\mathcal{P}$ be an RP-DeLP program with a single defeasibility level and let $(\mathit{Warr}, \mathit{Block})$ be the maximal ideal output for $\mathcal{P}$. Under this hypothesis, if $\Pi \cup \mathit{Warr} \vdash_R Q$, then $Q \in \mathit{Warr}$.*

The following example shows the closure result for the maximal ideal output.

Example 11. *Consider the RP-DeLP program $\mathcal{P}_{12} = (\Pi, \Delta, \preceq)$ with*

$$\Pi = \{\sim s \leftarrow q, \sim r \leftarrow h\} \text{ and } \Delta = \{q \leftarrow r, h \leftarrow s, r, s, q, h\},$$

and two defeasibility levels for Δ: α_1 and α_2 with $1 > \alpha_1 > \alpha_2 > 0$. Consider that Δ is stratified as follows:

$$\text{level } \alpha_1\text{: } \{q \leftarrow r, h \leftarrow s, r, s\} \qquad \text{level } \alpha_2\text{: } \{q, h\}.$$

Obviously, $\mathit{Warr}(1) = \emptyset$. Then, at level α_1, we have two valid arguments:

$$\mathcal{H}_1 = \langle \{r\}, r\rangle \text{ and } \mathcal{H}_2 = \langle \{s\}, s\rangle.$$

and four almost valid arguments with respect to $\{\mathcal{H}_1, \mathcal{H}_2\}$:

$$\mathcal{F}_1 = \langle \{r, q \leftarrow r\}, q\rangle, \quad \mathcal{F}_3 = \langle \{r, q \leftarrow r\}, \sim s\rangle,$$
$$\mathcal{F}_2 = \langle \{s, h \leftarrow s\}, h\rangle, \quad \mathcal{F}_4 = \langle \{s, h \leftarrow s\}, \sim r\rangle.$$

Figure 5.4 shows the warrant dependency graph for $\{\mathcal{H}_1, \mathcal{H}_2\}$ *and* $\{\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3, \mathcal{F}_4\}$*. The cycles express that either* r *or* s *can be warranted, but not both. Hence, at level* α_1*, we have two possible outputs for* $\mathcal{P}_{12}$*:*

$$\begin{array}{ll} Warr_1(\alpha_1) = \{r\}, & Block_1(\alpha_1) = \{s, q\}, \\ Warr_2(\alpha_1) = \{s\}, & Block_2(\alpha_1) = \{h, r\}. \end{array}$$

Then at level α_2*, all arguments are rejected in both outputs, and thus,* $Warr_1(\alpha_2) = Warr_2(\alpha_2) = \emptyset$ *and* $Block_1(\alpha_2) = Block_2(\alpha_2) = \emptyset$*. Therefore, the two possible outputs for* $\mathcal{P}_9$ *are:*

$$\begin{array}{ll} Warr_1 = \{r\}, & Block_1 = \{s, q\}, \\ Warr_2 = \{s\}, & Block_2 = \{h, r\}. \end{array}$$

Consider now the maximal ideal output for $\mathcal{P}_{12}$ *in which valid arguments involved in cycles are blocked and almost valid arguments involved in cycles are rejected. Obviously,* $Warr(1)_{maximal} = \emptyset$ *and, at level* α_1*, the maximal ideal output for* $\mathcal{P}_{12}$ *is:*

$$Warr_{maximal}(\alpha_1) = \emptyset, \qquad Block_{maximal}(\alpha_1) = \{r, s\}.$$

Now, at level α_2 *we have that arguments*

$$\langle\{q\}, q\rangle \text{ and } \langle\{h\}, h\rangle$$

are valid and none of them is involved in a cycle neither in a conflict, and thus, q *and* h *are warranted conclusions at level* α_2 *(i.e.* $\{q, h\} \subseteq Warr_{maximal}(\alpha_2)$*). Finally, although arguments*

$$\langle\{q, \sim s \leftarrow q\}, \sim s\rangle \text{ and } \langle\{h, \sim r \leftarrow h\}, \sim r\rangle$$

are recursively based on warranted conclusions, both violate Condition (V3) (i.e. $s, r \in Block_{maximal}(\geq \alpha_1)$*), and thus, both arguments are rejected since they are not valid. Hence, at level* α_2*,* s *and* r *are rejected for the maximal ideal output:*

$$Warr_{maximal}(\alpha_2) = \{q, h\}, \qquad Block_{maximal}(\alpha_2) = \emptyset.$$

Hence, the maximal ideal output for $\mathcal{P}_{12}$ *is:*

$$Warr_{maximal} = \{q, h\}, \qquad Block_{maximal} = \{r, s\}.$$

Therefore, due to the fact that the set of conclusions that are warranted and blocked at each level determines which arguments are valid at lower levels, we get that $\Pi \cup Warr_{maximal} \vdash_R \sim s$ *and* $\Pi \cup Warr_{maximal} \vdash_R \sim r$*, but* $\sim s, \sim r \notin Warr_{maximal}$ *since* $s, r \in Block_{maximal}(\alpha_1)$*.*

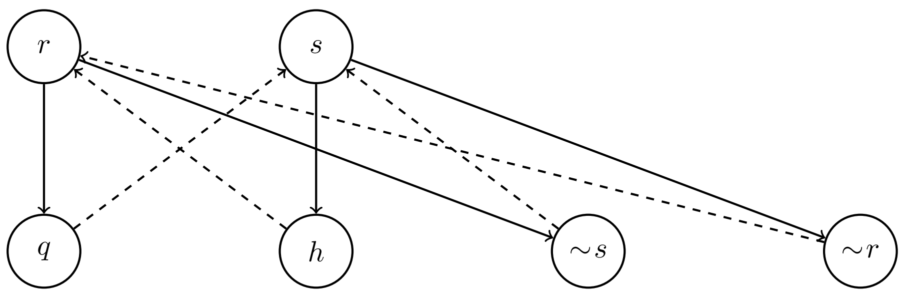


Figure 5.4: Warrant dependency graph for RP-DeLP program from example 11.

5.2. Computing the Maximal Ideal Output for an RP-DeLP Program

As in the computation of multiple outputs for an RP-DeLP program, the maximal ideal output of an RP-DeLP program can be computed by means of a level-wise procedure, as we have seen before starting from the highest level and iteratively going down from one level to next level below. It is necessary to determine the status (warranted or blocked) of each valid argument at every level. Next we show an algorithm which implements this level-wise procedure computing warranted and blocked conclusions by checking the existence of conflicts between arguments and cycles at some warrant dependency graph. As in the multiple output version we use the simpler notation W, $W(1)$, $W(\alpha)$ and $W(\geq \alpha)$ for *Warr*, *Warr*(1), *Warr*(α) and *Warr*($\geq \alpha$) respectively, and B, $B(\alpha)$ and $B(\geq \alpha)$ for *Block*, *Block*(α) and *Block*($\geq \alpha$), respectively.

```
Algorithm Computing_warranted_conclusions
Input $\mathcal{P} = (\Pi, \Delta, \preceq)$: An RP-DeLP program
Output $(W, B)$: maximal ideal output for $\mathcal{P}$
Method
    $W(1) := \{Q \mid \Pi \vdash_R Q\}$
    $B := \emptyset$
    $\alpha :=$ maximum_level($\Delta$)
    while ($\alpha > 0$) do
        level_computing($\alpha$, $W$, $B$)
        $\alpha :=$ next_level($\Delta$)
    end while
end algorithm
```

The main difference between the two algorithms is that now we won't have multiple extensions. One unique output is computed, so we don't need to stack partially computed outputs, as every warranted or blocked conclusion will be part of the same output.

The algorithm `Computing_warranted_conclusions` first computes the set of warranted conclusions $W(1)$ form the set of strict clauses Π. Then, for each defeasibility level $1 > \alpha > 0$, the procedure `level_computing` determines all warranted and blocked conclusions with strength α. Remark that for every level α, the procedure `level_computing` receives $W(> \alpha)$ and $B(> \alpha)$ as input and produces $W(\geq \alpha)$ and $B(\geq \alpha)$ as output.

```
Procedure level_computing (in $\alpha$; in_out $W$, $B$)
    $VL :\, = \{Q$ with strength $\alpha \mid \langle A, Q\rangle$ is valid w.r.t. $W$ and $B\}$;
    while ($VL \neq \emptyset$) do
      while ($\exists Q \in VL \mid \neg$ conflict($\alpha$, $Q$, $VL$, $W$,
      almost_valid_literals($\alpha$, $VL$, ($W$, $B$), $Q$)) do
            $W(\alpha) := W(\alpha) \cup \{Q\}$
            $VL :=\, VL \backslash \{Q\} \cup \{P$ with strength $\alpha \mid \langle C, P\rangle$
            is valid w.r.t. $W$ and $B\}$
      end while
      $I := \emptyset$
      if $VL$ does not change from previous iteration then
            $I :=\, VL$
      else
            for $Q \in VL$ do
                  if conflict($\alpha$, $Q$, $VL$, $W$, $\emptyset$) then
                        $I := I \cup Q$
                  end if
            end for
      end if
      $B(\alpha) := B(\alpha) \cup I$
      $VL := VL \backslash I$
    end while
end procedure
```

For any level α the procedure `level_computing` first computes the set VL of valid literals with respect to $W(> \alpha)$ and $B(> \alpha)$. Then, this set of valid literals is dynamically updated depending on new warranted and blocked conclusions with strength α. The procedure `level_computing` finishes when the status for every valid literals is computed.

We can see that computing the Maximal Ideal Output by means of the algorithm we described is easier than computing all the outputs of an RP-DeLP program with the algorithm described in Section 4.1. One can see that the process of solving a cyclic definition of a conflict by means of resolving the status of each member of the cycle, requires more computing time. Instead of that for the Maximal Ideal Output, when we detect a set of literals involved in at least one cycle, we block all of them, so the complexity is lower.

Observe that given the fact that queries `conflict` and `almost_valid _literal` have the same input and output variables that the procedures used in the algorithm described in Section 4.1, we can make use the same encodings described in Sections 4.2 and 4.3.

5.3. Web System Architecture

In this section we describe the architecture of the web based system we have designed and implemented to process RP-DeLP programs [1] and which is available at http://arinf.udl.cat/rp-delp.

A web application is a software tool which is available through a network of computers, in most of the cases this software is being executed in a server and accessible by all the clients connected to the same network. That software provides a common interface which is supported by a web browser. One of the main features of a web application is ubiquity, that means that the application is ready as long as there is access to the network. Another advantage is that the only software required at the client part is a web browser. A key reason for its popularity is the fact that the software can be updated and maintained without the necessity of installing or disturbing the clients. There are more advantages, such as the inherent cross-platform compatibility and the lack of high system requirements, because most of the processing tasks are executed in the server side.

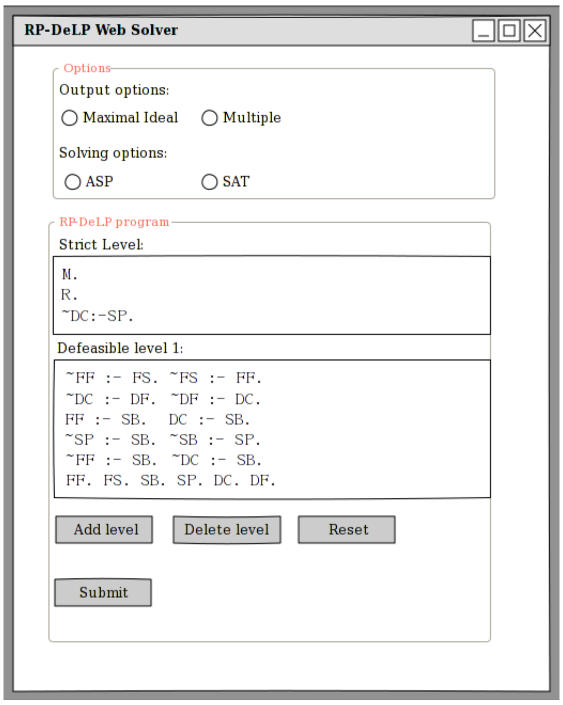
However, there are also some inconveniences, as a web interface is not easily adaptable to other systems. To solve this drawback we designed a web system capable of solving both instances posted through a web interface and also instances received through HTTP requests. By using HTTP protocol, any client will be able to send a query without the necessity of filling a web interface.

Our system is allocated in a stand-alone server where all the required software is installed, including a web server to handle HTTP requests, the RP-DeLP algorithm to handle the user programs and the rest of the software required by the system, such as the ASP and SAT solvers. As said previously, the user can access to the server using a web browser through the web interface or by posting an HTTP request. The RP-DeLP algorithm is implemented with Python. In Figure 5.5 we show the web interface structure of the system.

The web interface is divided into two main parts. The first part is devoted to choose the computation options. The *Maximal Ideal* radio button computes the maximal ideal output and the *Multiple* radio button computes the set of outputs. Remark that for RP-DeLP programs with a single output both options will compute the same output.

The *SAT* radio button encodes queries with SAT formulas and solves them with solver Minisat solver. The *ASP* radio button encodes queries

[1]We plan to deliver an open-source version as soon as we consider the system is sufficiently mature.



```
M.
R.
~DC:-SP.
```

```
~FF :- FS. ~FS :- FF.
~DC :- DF. ~DF :- DC.
FF :- SB.  DC :- SB.
~SP :- SB. ~SB :- SP.
~FF :- SB. ~DC :- SB.
FF. FS. SB. SP. DC. DF.
```

Figure 5.5: Web interface form for submitting RP-DeLP programs.

with ASP formulas and solves them with solver Clingo.

The second part of the web interface is devoted to define RP-DeLP programs. Each defeasible level is depicted by a text box. The strict part of the program is defined in the *Strict Level* text box. The defeasible part of the program can be defined using multiple levels of strength, starting from the highest level and going down from one level to the level below. The form is dynamically updated, so the user can add new levels by pressing the *Add level* button. The *Delete level* erases the last added level and the *Reset* button deletes all levels except the strict part and the first defeasible level. The *Submit* button starts the computation process.

Clauses are defined as follows:

- All clauses must end with a dot.
- A literal is an alphanumeric word starting with a letter.
- Negation is denoted with symbol $\sim$.
- Implication symbol is written with `:-` and conjunction symbol is a comma. For instance, $q \leftarrow p_1 \wedge \sim p_2$ is written in our formalism as: `q :- p1, ~p2`.

Regarding the system architecture, represented in Figure 5.6, the computation process starts with the translation of the RP-DeLP program and the set of computation options into an XML file. Then, appropriate Python structures are built from the XML file in order to be processed by the RP-DeLP algorithm. The RP-DeLP algorithm uses the ASP or the SAT implementation depending on the computation options. When the RP-DeLP algorithm finishes, outputs are stored in an XML file which allow us to provide an HTTP response or an HTML page.

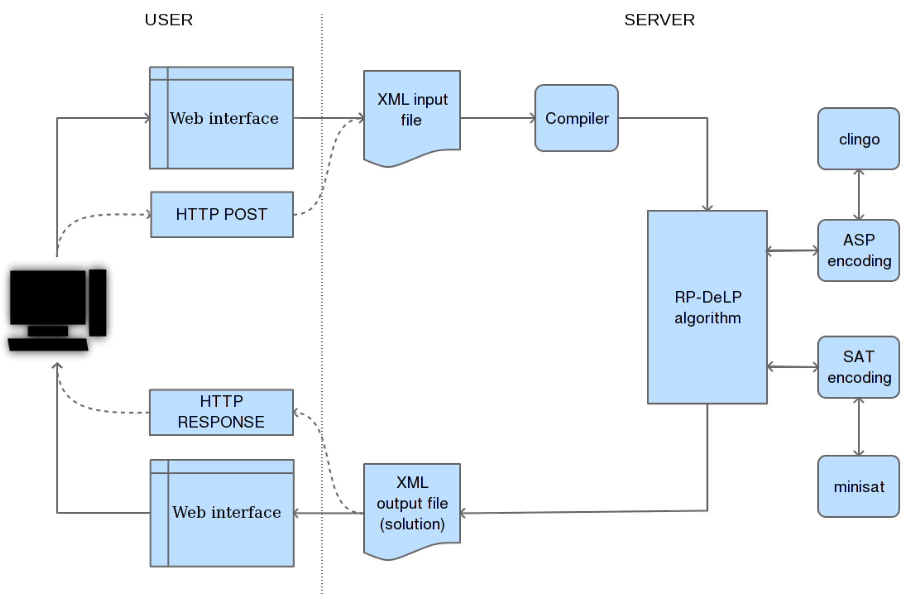


Figure 5.6: RP-DeLP web system architecture.

One of the main features of our system is that provides not only sets of warranted and blocked conclusions, but also information to further understand the reasons why conclusions are warranted or blocked.

In Table 5.1 we show the information that the system provides to the user for every output. This information is divided in two parts. The first part shows the total number of outputs, a label of the computing order of the output, the set of warranted conclusions of the output and the time expended computing the output. For each warranted conclusion there is a list of conclusions which support it and the strength of the conclusion. The second part shows the set of blocked conclusions of the output and the set of conclusions which support them. For each blocked conclusion the system informs about the strength of the conclusion and the reason that leads to block it: a conflict or a cycle. For conflicts it shows the set of valid conclusions that minimally conflicts. For cycles it shows the set of valid conclusions of the cycle at some warrant dependency graph. Remark that a conclusion is blocked due to a cycle whenever we compute the maximal ideal output, otherwise conclusions are blocked due to conflicts.

Number of outputs	# output	Warranted conclusions	Time
n	i	$P_1 : \{P_{1,1}, \ldots, P_{1,h}\}\ [\alpha_{P_1}]$	t_n
		$\vdots$	
		$P_r : \{P_{r,1}, \ldots, P_{r,u}\}\ [\alpha_{P_r}]$	

# output	Blocked conclusions	Support of blocked conclusions
i	$Q_1 : \mathit{conflict}(Q_{1,1}, \ldots, Q_{1,j})\ [\alpha_{Q_1}]$	$Q_1 : \{P_{1,1}, \ldots, P_{1,k}\}$
	$\vdots$	$\vdots$
	$Q_s : \mathit{cycle}(Q_{s,1}, \ldots, Q_{s,v})\ [\alpha_{Qs}]$	$Q_x : \{P_{x,1}, \ldots, P_{x,w}\}$

Table 5.1: Information provided by the system for an RP-DeLP program.

5.4. A Running Example: Arguing about the Best Menu

In this section we consider the application of the RP-DeLP argumentation framework to the construction of suitable menus in a restaurant. Suppose we have two persons in the restaurant arguing about how to select the different menu items: Chicote (the chef) and Luis (the restaurant manager). Chicote is more concerned about the quality of the menu, whereas Luis is more concerned about the price of the menu. However, both agree that for preparing the menu they should reach a consensus that considers the preferences of both.

The menu must contain appetizer, drink, first course, second course and dessert. For appetizer and drink they have already reach the conclusion that they will serve mussels (M) and red wine (R), respectively. But there is no a consensus about the other items.

For the first course, the options are Soup (FS) and Fish (FF). For the second course are Beef (SB) and pork (SP). And for the dessert are fruit (DF) and Sacher cake (DC).

First case. As we have said, they both agree on the selection for the appetizer and the drink. Also, they both agree that when pork is served, Sacher cake cannot be served. So, at the strict level we have the following hard constraints:

$$\Pi \;=\; \{M, R, {\sim}DC \leftarrow SP\}$$

The other conditions for the menu are not so clear, as both sides have some opposite preferences, or not any preferences between some options. First, regarding what option to select for each course and dessert, they do not have, a priori, any preference between the two options for each menu item. But it is clear that once one option is selected, the other should be avoided. So, at the defeasible level we have propositions for all the possible options for first and second course and dessert and rules that express the preference that once one option is selected for a course or for the dessert, the other should not be selected.

Secondly, regarding the preferred combinations of courses and dessert, Luis prefers not to serve the more expensive first course (fish) and the more expensive dessert (Sacher cake) when beef is the second course. By contrast, Chicote believes that when beef is served, the preferred options for first course and dessert are fish and Sacher cake.

Then, all these conditions are encoded with a single defeasible level as follows: [2]

$$
\begin{array}{lll}
\Delta_{\alpha_1} & = & FF, FS, SB, SP, DC, DF, \\
& \{ & \\
& & F1 : {\sim}FF \leftarrow FS, \qquad F2 : {\sim}FS \leftarrow FF, \\
& & S1 : {\sim}SP \leftarrow SB, \qquad S2 : {\sim}SB \leftarrow SP, \\
& & D1 : {\sim}DC \leftarrow DF, \qquad D2 : {\sim}DF \leftarrow DC, \\
& & C1 : FF \leftarrow SB, \qquad C2 : DC \leftarrow SB, \\
& & L1 : {\sim}FF \leftarrow SB, \qquad L2 : {\sim}DC \leftarrow SB \ \}
\end{array}
$$

Given all these conditions, it turns out that there is an unique menu that we extract with our argumentation system, that corresponds with the unique output of our program shown in Table 5.2.

# of outputs	Warranted conclusions	Blocked conclusions	Support of blocked
1	$M : \{M\}[\Pi]$ $R : \{R\}[\Pi]$ $SB : \{SB\}[\alpha_1]$ $DF : \{DF\}[\alpha_1]$ $FS : \{FS\}[\alpha_1]$	$SP : conflict(SP, DC)[\alpha_1]$ $DC : conflict(SP, DC)[\alpha_1]$ $FF : conflict(FF, {\sim}FF)[\alpha_1]$ ${\sim}FF : conflict(FF, {\sim}FF)[\alpha_1]$	${\sim}FF : \{SB\}$ $FF : \{FF\}$ $SP : \{SP\}$ $DC : \{DC\}$

Table 5.2: Output of the system for our first running example.

The reasons for this unique output are as follows. First, observe that there is a conflict between valid arguments for SP and DC, so they are blocked and SB and DF can be warranted. Then given the warrant status of SB and the defeasible rule $L1$, ${\sim}FF$ becomes valid, but then there is a conflict between FF and ${\sim}FF$, so they are blocked. This allows to warrant FS. As we have an unique output, in this case this output coincides with the maximal ideal output of the program.

Second case. Suppose now that after some deliberation between Luis and Chicote, they agree that the preference of Chicote of having fish and cake when we have beef should receive more consideration than the preference of Luis of not having fish and cake. But we still do not have a preference between fish and soup, so they are still in the same defeasible level. So, we have the same strict knowledge as before, but two defeasible levels α_1 and α_2 with $\alpha_1 > \alpha_2$. Then, the set of defeasible facts and rules is stratified as

[2]The RP-DeLP program of Figure 5.5 corresponds with the set of facts and rules of this example.

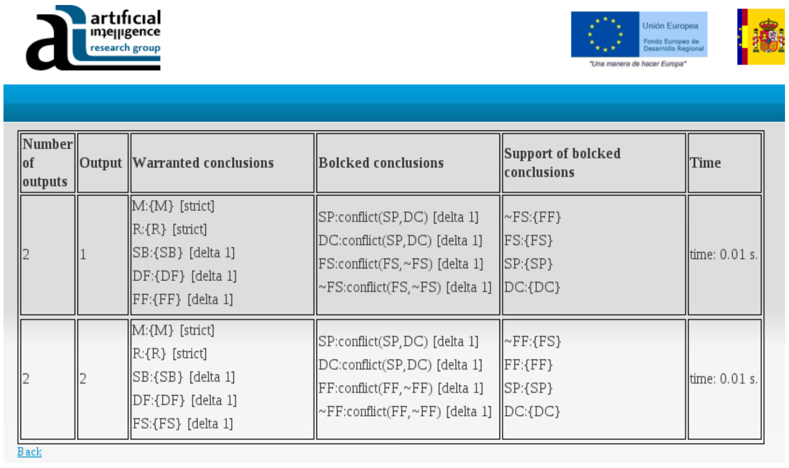
follows:

$$
\begin{array}{rcll}
\Delta_{\alpha_1} & = & FF, FS, SB, SP, DC, DF & \\
 & \{ & & \\
 & & F1 : \sim FF \leftarrow FS, & F2 : \sim FS \leftarrow FF, \\
 & & S1 : \sim SP \leftarrow SB, & S2 : \sim SB \leftarrow SP, \\
 & & D1 : \sim DC \leftarrow DF, & D2 : \sim DF \leftarrow DC, \\
 & & C1 : FF \leftarrow SB, & C2 : DC \leftarrow SB \ \} \\
 & & & \\
\Delta_{\alpha_2} & = & L1 : \sim FF \leftarrow SB, & L2 : \sim DC \leftarrow SB \ \} \\
 & \{ & &
\end{array}
$$

Given the modified defeasible knowledge, we have that now two menus are possible, that correspond with the two outputs we have this time. We have two outputs because now the warrant status of SB does not create a conflict between FF and $\sim FF$. So, as both FF and FS are valid arguments at the defeasible level α_1, together with the defeasible rules $F1$ and $F2$ we have two conflicts with almost valid arguments that cannot be resolved because there is a cyclic dependence. To break this cycle, we have to consider two options: either to warrant FF or to warrant FS:

1. If we warrant FF, then $\sim FS$ becomes valid so we have to block FS and $\sim FS$. This gives our first output in Figure 5.7.

2. If we warrant FS, then $\sim FF$ becomes valid so we have to block FF and $\sim FF$. This gives our second output in Figure 5.7.


artificial intelligence research group
Unión Europea
Fondo Europeo de Desarrollo Regional
"Una manera de hacer Europa"
Back

Number of outputs	Output	Warranted conclusions	Bolcked conclusions	Support of bolcked conclusions	Time

2	1	M:{M} [strict]
		R:{R} [strict]
		SB:{SB} [delta 1]
		DF:{DF} [delta 1]
		FF:{FF} [delta 1]
			SP:conflict(SP,DC) [delta 1]
			DC:conflict(SP,DC) [delta 1]
			FS:conflict(FS,~FS) [delta 1]
			~FS:conflict(FS,~FS) [delta 1]
				~FS:{FF}
				FS:{FS}
				SP:{SP}
				DC:{DC}
					time: 0.01 s.

2	2	M:{M} [strict]
		R:{R} [strict]
		SB:{SB} [delta 1]
		DF:{DF} [delta 1]
		FS:{FS} [delta 1]
			SP:conflict(SP,DC) [delta 1]
			DC:conflict(SP,DC) [delta 1]
			FF:conflict(FF,~FF) [delta 1]
			~FF:conflict(FF,~FF) [delta 1]
				~FF:{FS}
				FF:{FF}
				SP:{SP}
				DC:{DC}
					time: 0.01 s.

Figure 5.7: Output of the system for our second running example.

For this case, the maximal ideal output has content shown in Table 5.3. Remark that for our example the set of warranted conclusions for the maximal ideal output coincides with the intersection of the set of warranted conclusions for each output which indicates that Chicote and Luis coincide always at least on the second course and the dessert. [3]

# of outputs	Warranted conclusions	Blocked conclusions	Support of blocked
1	$M : \{M\}[\Pi]$	$SP : conflict(SP, DC)[\alpha_1]$	$FF : \{FF\}$
	$R : \{R\}[\Pi]$	$DC : conflict(SP, DC)[\alpha_1]$	$FS : \{FS\}$
	$SB : \{SB\}[\alpha_1]$	$FF : cycle(FF, FS)[\alpha_1]$	$SP : \{SP\}$
	$DF : \{DF\}[\alpha_1]$	$FS : cycle(FF, FS)[\alpha_1]$	$DC : \{DC\}$

Table 5.3: Maximal ideal output for the second running example.

[3]The set of warranted conclusions for the maximal ideal output is not equal to the intersection of the set of warranted conclusions for each output for all RP-DeLP program with multiple outputs. However, the set of blocked conclusions for the maximal ideal output is different to the intersection of the set of warranted conclusions for each output for all RP-DeLP program with multiple outputs.

6 Average Computational Cost and Easy/Hard Problem Instances

To study the scaling behavior of the computational cost of our algorithms as the size increases, as well as how different characteristics of the problem instances affect its computational cost, we have implemented our algorithms and conducted a series of experiments.

The main structures of algorithms have been implemented with the programming language Python, but for solving the SAT and ASP formulas presented in the previous section we used different solvers. For the SAT version of the algorithm, it uses Minisat SAT solver. However, our architecture easily allows to use any other SAT solver that appears in the future. Minisat is one of the publicly available SAT solvers which implements most of the current state-of-the-art solving techniques such as conflict-clause recording and conflict-driven backjumping, among others. When using the ASP encodings the algorithm uses Clingo and other tools from Potassco suite [GKK+11b]. Potassco suite is the best performing ASP solver in the last ASP competition [CIR+11].

In the experiments the algorithm solves different test sets of problem instances obtained with a random generation algorithm. The instances can be solved using the SAT solver or the ASP solver, as we have presented two encoding based on SAT and ASP formulas. Also, for each problem instance we can calculate all the outputs or the maximal ideal output.

To study and analyze how our RP-DeLP algorithm behaves as different characteristics of the problem change, we generated our instances using one and two levels of defeasibility and changing the other parameters of the problem instances. The experiments have been performed solving different test sets of the randomly generated problem instances and have been run on machines with the following specs: Rocks Cluster 5.2 Linux 2.6.18 Operating System, AMD Opteron 248 Processor clocked at 1GHz, 1.0GB Memory, and GCC 4.1.2 Compiler.

6.1. Random Generation of RP-DeLP Problem Instances

We used different parameters to control the generation of random RP-DeLP problem instances with different sizes, defeasibility levels and other characteristics. To experiment with different defeasible levels we generated for our experimentation first on one set of problems with only one defeasible level and then on another set with two defeasible levels. In both cases we were interested in how the resolution time differs when the ratio of clauses to the number of variables increases. Then, in the first case with only one defeasible level, we were also interested in the results when the fraction of clauses of the program at the strict knowledge level is modified, ranging from no strict knowledge at all to all clauses at the strict knowledge level. For the case of two defeasible levels, we have investigated the effect of modifying the fraction of clauses between the two defeasible levels. We next explain the generation of our problem instances.

- **Generation of instances with one defeasible level.** Given a number of variables (V), a maximum clause length (ML), a ratio of clauses to variables (C/V), and a fraction (f), between 0.0 an 1.0, of strict knowledge, the algorithm generates an RP-DeLP problem instance by generating C clauses, such that the length of the body of every clause is selected uniformly at random from $[0, ML]$ (clauses with body length 0 are facts). The variables of the literals of a clause are selected uniformly at random without repetition, and are negated with probability 0.5. From the C clauses, $f \cdot C$ clauses are in the strict knowledge and the rest in the defeasible set.

- **Two defeasible levels instance generation.** Similar to the previous instance generator with a number of variables (V), a maximum clause length (ML), a ratio of clauses to variables (C/V), now we fix the fraction of strict knowledge (f) to some value. Then two defeasible levels are built assigning a fraction l between 0.0 and 1.0 of the total number of defeasible clauses to the first defeasible level and 1-l to the second defeasible level.

6.2. Maximal Ideal Output

6.2.1. Test instances considered

To perform the empirical analysis for the maximal ideal algorithm we generated used two different groups of test sets: test sets with one defeasible level and test sets with two defeasible levels. In both groups, test instances were created with a number of variables (V) selected from $\{20, 25, 30, 35\}$, [1] and with maximum clause length (ML) selected from $\{2, 4\}$.

In the case of one defeasible level, for each combination (V, ML), different test sets of instances were created by selecting a number of total clauses, such that the ratio C/V ranged from 1 to 14 in steps of 0.5, and the fraction of clauses in the strict knowledge ranged from 0 to 0.9 in steps of 0.1. So, the total number of test sets for each combination (V, ML) was 90. The number of instances generated in each test set was 50.

In the case of two defeasible levels, for each combination (V, ML) and an strict knowledge fraction set to 0.1, different test sets of instances were created by selecting a number of total clauses, such that the ratio C/V ranged from 1 to 14 in steps of 0.5, and the fraction of clauses in the first level l ranged from 0.1 to 0.9 in steps of 0.1.

6.2.2. One defeasible level

For one defeasible level, we first show the results for instances with total number of variables $V = 30$ and maximum clause length $ML = 2$. The left plot of Figure 6.1 shows the median time to solve the instances with the SAT approach when solving instances with different ratio of the number of total clauses to number of variables (axis labelled with C/V in the plots) and with different fraction of strict knowledge (axis labelled with *strict knowledge*). The plot shows that for a strict knowledge fraction of 0.0, there is an increase of the median time as the total number of clauses increases.

By contrast, as the strict knowledge fraction increases, the time increases only up to certain value of the number of total clauses, and then drops significantly. This is probably because of two causes:

- the more strict knowledge we have, the more possibilities to have inconsistent instances, that are detected in polynomial time by our algorithm, and
- the more unacceptable arguments and blocked literals we can have.

To check the possible role of inconsistent instances on the complexity of the problem, we have also computed what fraction of the instances, for each test set of 50 instances, are inconsistent ($\Pi \vdash_R \perp$). The right plot of Figure 6.1

[1]Notice that the total number of literals will be two times the number of variables.

shows this information. The color scale ranges from points with a fraction of instances with inconsistent strict knowledge equal to 0 (dark blue color) to points with such fraction equal to 1.0 (red color). Apart for the obvious case of strict knowledge fraction equal to 0.0, where there are never inconsistent instances, for a fraction of strict knowledge equal to 0.1 up to the ratio $C/V = 6$ no inconsistent strict knowledge is generated, but the time needed to solve the instances is smaller than the one needed for instances with no strict knowledge at all.

As the fraction of strict knowledge increases, test instances with inconsistent strict knowledge appear more frequently for a lower ratio C/V and the interval of values of C/V with instances with significant computation time (greater than 0) decreases. Also, the highest computation time obtained decreases as the fraction of strict knowledge increases.

To further understand the reasons for such differences on the computation time, we have also studied the average ratio of warranted literals and average ratio of blocked literals, with respect to the total number of variables, for each test set. The left plot of Figure 6.2 shows the ratio of warranted literals and the right plot the ratio of blocked literals. Looking at both plots, we observe that for instances with low C/V, if its strict knowledge fraction is also low, we have a small, but non-negligible, fraction of warranted literals, that starts to increase as we increase C/V, but only up to certain limit C/V (around 2.0), and above that limit the fraction of warranted literals starts to decrease, coinciding with an increase in the fraction of blocked literals. A plausible explanation for this is that for very low C/V instances have very few valid arguments, so few warranted and blocked literals are produced. As C/V increases, more valid arguments start to appear, but obviously as the number of valid arguments increases more and more of them will be part of a conflict set of arguments. So, it seems that the highest computation times are found for instances with enough clauses such that many valid arguments are found, but many of them are also found to be part of a collective conflict set.

When the strict knowledge fraction increases, as the fraction of instances with inconsistent strict knowledge increases, it is clear that on average warranted and blocked literals will decrease, and this is observed on both plots. It is also natural that even on instances with a consistent strict knowledge, when this fraction is larger, less literals will have valid arguments, because consistency with the strict knowledge will hold for less arguments. However, we still find remarkable the increase of easy instances for a strict knowledge fraction of only 0.2, because at this strict knowledge fraction for C/V up to 5.0 instances still have warranted literals. A possible explanation for this increase of easy instances even when we still have a considerable number of warranted literals, is that the fraction of strict knowledge produces the pruning of larger arguments, so the arguments found for warranted literals are shorter and easier to find.

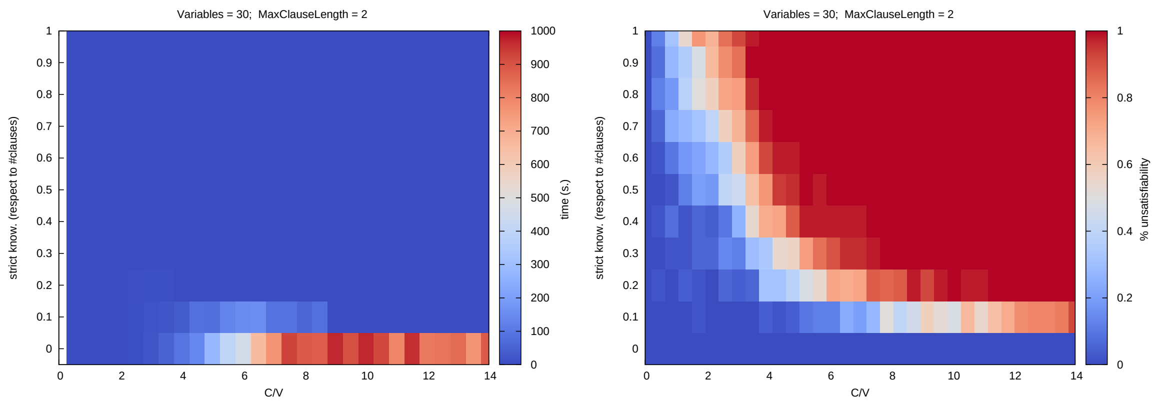


Figure 6.1: Median time to solve the instances (left) and fraction of inconsistent instances (right) for $V = 30, ML = 2$.

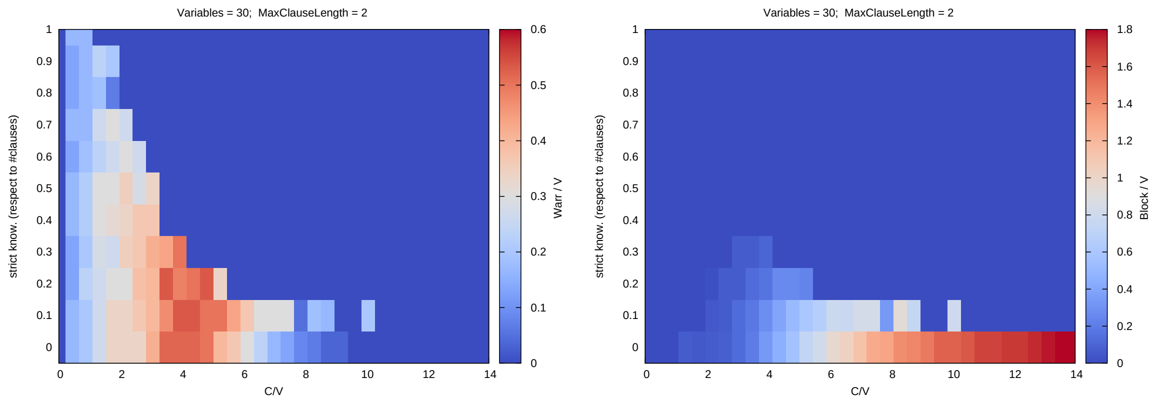


Figure 6.2: Warranted literals (left) and blocked literals (right) for $V = 30, ML = 2$.

Our next goal is to compare our two solving approaches for the maximal ideal output, we want to run a set of instances with ASP and SAT approaches in order to compare the time expended in solving such instances. To do so, we will avoid using unsatisfiable instances as the solver will find an inconsistency quickly and there will be no calls to any solver. So one of our sets will be the set with instances without strict knowledge (this is the set of instances with fraction of strict knowledge f equal to 0), as in this set there will not include any inconsistent instance.

We will also choose the set with fraction of strict knowledge $f = 0.1$. Despite of it will surely contain some inconsistent instances, results show that the inconsistency ratio is still low in comparison with the rest of the sets with higher number of clauses in the strict part (this is instances with fraction of strict knowledge $f < 0.1$. We believe that some instances in this set are challenging because require a high number of solver calls in order to find collective conflicts.

Next we show a comparison in the average computational cost of the two SAT and ASP approaches. In Table 6.1, we compared the time expended in computing the maximal ideal output of instances with $V = 30, ML = 2$ and fraction of strict knowledge $f = 0.0$ and $f = 0.1$. Results show that the ASP approach outperforms the SAT approach. One reason for the better performance of ASP can be that ASP encodings are smaller than SAT ones and thus the search space is bigger in SAT encondigns. As ASP and RP-DeLP both inherit its language from Logic Programming, rules and facts can be directly encoded from RP-DeLP to ASP. By contrast, in SAT it is needed to explicitly encode the non-deterministic choice of clauses for inferring conclusions in an argument. It is also worth to note that following the idea of SAT encoding for classical STRIPS problems, each rule and each fact must be rencoded in each level of the plan, adding also linking rules between levels. This is not needed in the ASP encoding, so ASP instances tend to be smaller.

In Table 6.2 we show the hardest instances for each set with different V and a fixed number of max clause length with $ML = 2$. We compare both approaches ASP and SAT by showing the time to solve the hardest instances and also we show the fraction C/V of the instances. Again we see how ASP needs less time to compute the maximal ideal output for the given instances.

In Table 6.3 we show the same results but now for a maximum clause length of $ML = 4$.

6.2.3. Two defeasible levels

Next, we analyze the effect on complexity of having two defeasible levels, instead of just one. For these instances we have fixed the strict knowledge fraction to 0.1 because we wanted to test the hardest possible instances we can have when there is a fraction of strict knowledge greater than zero, so we still can have non-trivial conflicts between arguments due to the role of the strict knowledge on collective conflicts.

We first take a look at the results obtained after solving the set of instances with the number of variables $V = 30$ and maximum clause length $ML = 2$. The left plot of Figure 6.3 shows the median time to solve the instances with our algorithm when solving instances with different ratio of the number of total clauses to number of variables (axis labeled with C/V in the plots) and with different fraction of defeasible knowledge at the first defeasible level (axis labeled with *fraction l*). The right plot of the same figure shows the percentage of consistent instances. We observe that as before, just up to the ratio where almost all instances are inconsistent, there is an increase on the median time.

However, the lowest computation times are found on a range of values

	SAT		ASP	
	f		f	
C/V	0.0	0.1	0.0	0.1
4.0	225.91	103.66	42.66	34.85
5.0	480.51	180.65	52.86	34.21
6.0	598.55	288.35	60.95	37.60
7.0	756.08	199.27	69.36	33.96
8.0	855.64	172.78	64.90	33.96
9.0	913.16	79.03	67.03	33.04
10.0	967.87	59.03	66.49	27.66
11.0	787.51	45.15	65.17	21.03
12.0	779.84	38.15	63.00	22.26
13.0	829.43	32.07	63.57	16.13
14.0	892.32	28.12	52.79	14.04

Table 6.1: Time in seconds to solve random instances with $V = 30, ML = 2$ and fraction of strict knowledge $f = 0.0$ and $f = 0.1$ with SAT and ASP encoding approaches.

V	15		20		25		30		35	
	C/V	time	C/V	time	C/V	time	C/V	time	C/V	time
ASP	8.0	10.02	8.0	27.68	9.0	47.84	7.0	69.36	11.0	140.02
SAT	9.0	64.03	10.0	229.08	11.0	522.36	10.0	966.86	10.0	1500.76

Table 6.2: Fraction of C/V and time to solve (seconds) hardest instances with $ML = 2$ and fraction of strict knowledge $sk = 0.0$ with SAT and ASP encoding approaches.

V	15		20		25		30		35	
	C/V	time	C/V	time	C/V	time	C/V	time	C/V	time
ASP	11.0	14.80	8.0	28.83	11.0	63.51	9.0	123.83	12.0	172.91
SAT	13.0	119.79	13.0	341.23	9.0	725.39	11.0	1106.47	12.0	1224.20

Table 6.3: Fraction of C/V and time to solve (seconds) hardest instances with $ML = 4$ and fraction of strict knowledge $f = 0.0$ with SAT and ASP encoding approaches.

for the first level fraction around 0.5, and where this fraction is near 0 or 1 the computation time increases. A possible explanation for this concentration of the hardest instances when the defeasible knowledge is unbalanced (concentrated almost in one level) may be the following.

When almost all the clauses are in one level, we have more possible acceptable arguments in that level. Then, the space of possible collective conflicts at that level is also larger, so the computation times for the conflict queries will be higher. However, there is an slight difference in the computational cost when ($l \approx 0$) and when ($l \approx 1$). Despite in both cases we have the same unbalance of clauses between levels, having ($l \approx 0$) means that the contribution to the output of the program due to the first level will be small and quickly computed. At the second level, where the input will include the warrants from the strict part plus some warrants obtained from the first defeasible level, we will have less possible acceptable arguments from the second level than we would have if the first defeasible level would be empty. So, the total computational effort should be smaller than if all the defeasible knowledge would be only at one level. When we have the situation where ($l \approx 1$), almost all the defeasible knowledge is at the first level, so the computational effort to compute the output of the first level increases. That is, the input for the first defeasible level will contain only the warrants from the strict part, so the set of possible acceptable arguments will be larger (compared with the second defeasible level when ($l \approx 0$)) and the set of possible warrants and blocked literals to check will be larger. Observe that the plot for blocked literals at the right of Figure 6.4, shows a larger ratio $|B|/V$ for $l = 0.9$, as the number of clauses increases, than for $l = 0.1$.

So, when the fraction of clauses at the two defeasible levels is near 0.5, the number of warrants obtained from the first defeasible level will increase with respect to $l = 0.1$, but the number of blocked literals will be smaller than for $l = 0.9$, because the number of clauses at the first defeasible level is smaller. At the second defeasible level, the warrants and blocked literals will decrease, with respect to the case $l = 0.1$, given the input from the previous level. Looking at the left plot of Figure 6.4, that shows the ratio of warranted literals and the right plot the ratio of blocked literals, we clearly observe that more warranted literals are obtained around $l = 0.5$ but less blocked literals than at the extreme values of l.

Those results show that when defeasible levels are balanced in terms of number of clauses, there are less conflicts between arguments at the same level. That means that more literals can be warranted, and as it has been shown the lack of conflicts decreases the computation time of the output.

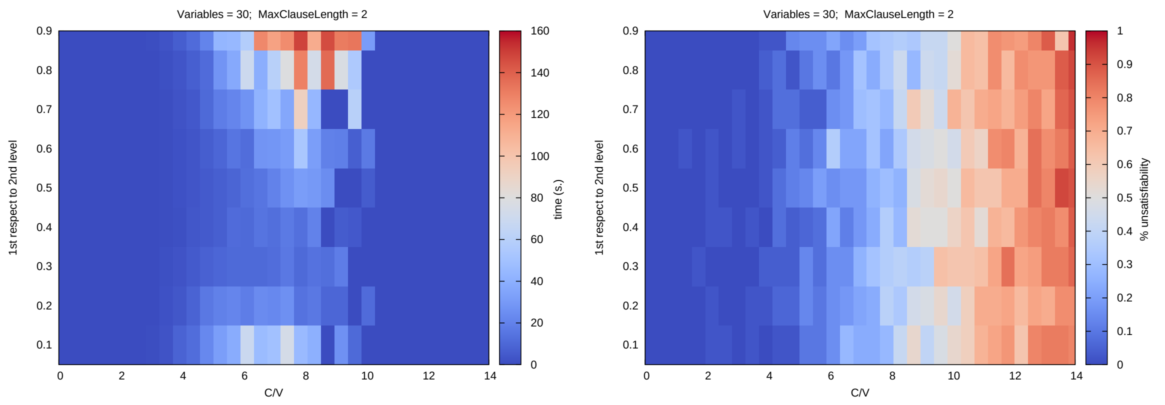


Figure 6.3: Average computational cost (left) and fraction of inconsistent instances (right) for $V = 30, ML = 2$, fraction of strict knowledge $= 0.1$ and two defeasible levels.

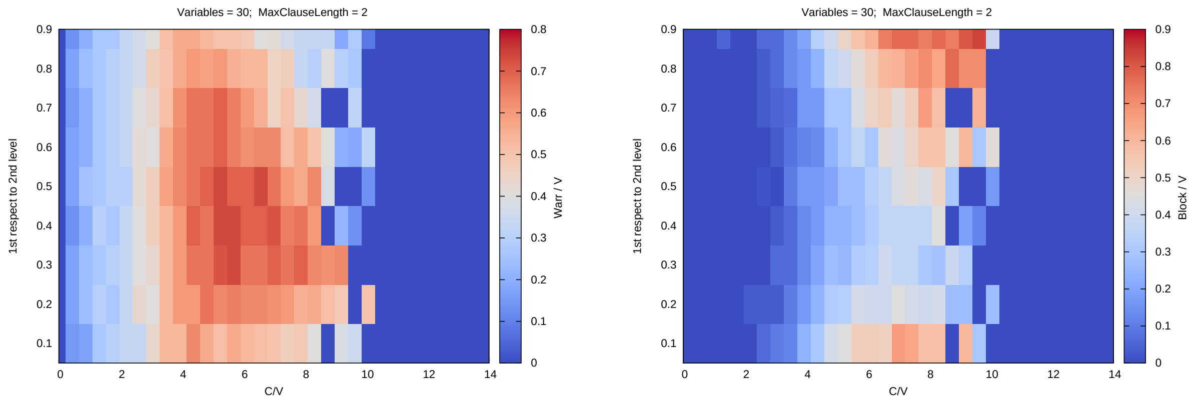


Figure 6.4: Warranted literals (left) and blocked literals (right) for $V = 30, ML = 2$, fraction of strict knowledge $= 0.1$ and two defeasible levels.

6.3. Multiple Outputs

To study the performance of the multiple outputs algorithm we used the same test instances used in Section 6.2.1. According with the previously presented results, we have shown that the approach using ASP encodings performs much better than the SAT approach. For this reason in this section we will use the ASP approach in our empirical tests.

Our first goal is to study the computational cost of solving the test instances. In Figures 6.5 and 6.6 we show the execution time to solve the test instances with $ML = 2$ using the ASP approach.

The first thing to note respect to the results shown in Figures 6.1 and 6.2 is that the maximum solving times are greater than the solving times for the maximal ideal output algorithm. One possible reason for these higher

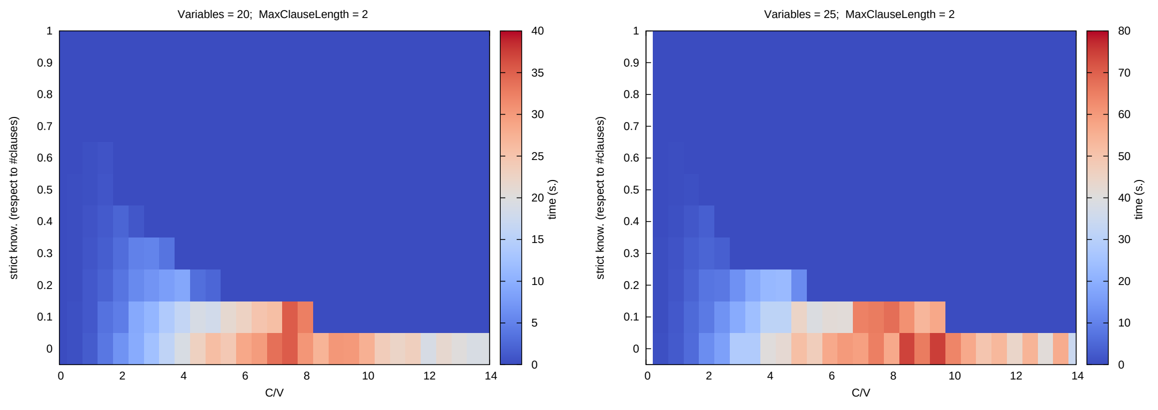


Figure 6.5: Computing time for $V = 20, ML = 2$ (left) and for $V = 25, ML = 2$ (right).

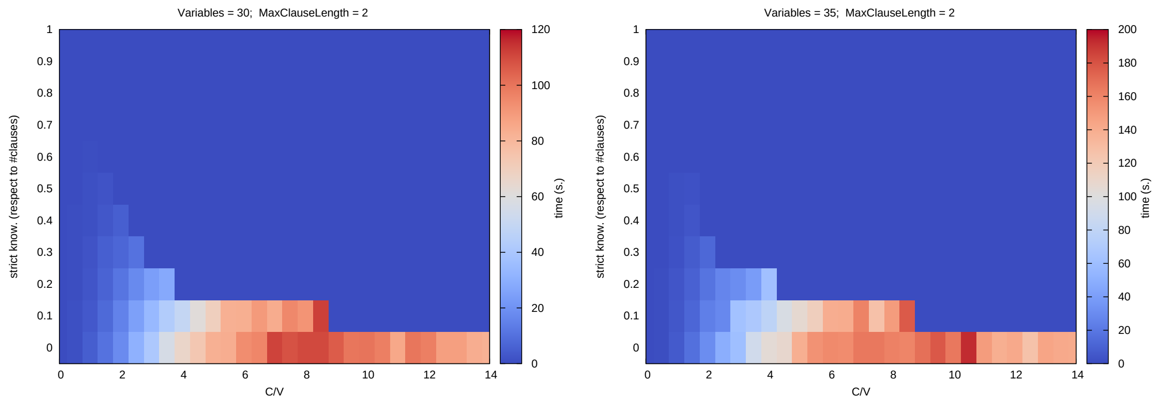


Figure 6.6: Computing time for $V = 30, ML = 2$ (left) and for $V = 35, ML = 2$ (right).

solving times could be the existence of instances with two or more outputs to compute, given that this would increase the total execution time of the algorithm. For this reason we next analyze the number and size of the outputs of the test instances when solved by the multiple outputs algorithm.

Table 6.4 shows the mean solving times and the mean number of outputs for the instances with $f = 0.0$ of the test sets considered in Figures 6.5 and 6.6 and Table 6.5 shows the main characteristics of these outputs. We focus on the instances with $f = 0.0$ because they give the hardest solving times. The characteristics we study for the outputs of the instances of each test set are: mean number of warrant literals divided by V (Mean Warr.) and mean number of blocked literals divided by $2V$ (Mean Block.). We observe that the maximum solving times are obtained for the C/V ratios that give instances with a higher mean number of outputs, and that around these ratios with highest solving times and highest number of outputs these quantities seem

V	C/V	f	time	#O
	6.0	0.0	28.55	1.42
	6.5	0.0	29.85	1.50
20	7.0	0.0	33.68	1.63
	7.5	0.0	35.18	2.00
	8.0	0.0	30.43	1.36
	8.5	0.0	27.05	1.14
	7.5	0.0	64.69	1.29
	8.0	0.0	56.57	1.05
25	8.5	0.0	74.09	1.80
	9.0	0.0	65.55	1.21
	9.5	0.0	74.98	1.82
	10.0	0.0	63.92	1.1
	6.5	0.0	95.55	1.17
	7.0	0.0	111.42	1.56
30	7.5	0.0	107.85	1.20
	8.0	0.0	101.71	1.16
	8.5	0.0	110.05	1.44
	9.0	0.0	105.67	1.04
	9.5	0.0	177.38	1.05
	10.0	0.0	164.87	1.36
35	10.5	0.0	192.23	1.49
	11.0	0.0	149.09	1.07
	11.5	0.0	138.90	1.05
	12.0	0.0	142.01	1.05

Table 6.4: Time to solve (seconds) and average number of outputs for the hardest instances with $C/V = \{20, 25, 30, 35\}$ and fraction of strict knowledge $f = 0.0$ with ASP encoding approach.

to be lower.

Looking at the characteristics of the outputs in Table 6.5 for a wider range of C/V ratios, we observe that the mean number of warranted literals increases before we reach the C/V ratio with the highest solving times, and then it starts to decrease. By contrast, the mean number of blocked literals increases monotonically almost always. A possible explanation for this behaviour is that when the number of clauses is low we have few valid arguments so we have few warranted and blocked literals. As the number of clauses increases, more warranted and blocked literals start to appear and also cyclic dependencies between almost valid and valid arguments that are the source for possible different outputs. When more clauses are added, more blocked literals present in any output will appear, given the presence

	Mean Warr.				Mean Block.			
	V				V			
C/V	20	25	30	35	20	25	30	35
1.0	0.31	0.25	0.26	0.25	0.02	0.03	0.02	0.03
2.0	0.44	0.42	0.39	0.43	0.12	0.11	0.13	0.12
3.0	0.48	0.51	0.50	0.50	0.25	0.23	0.23	0.26
4.0	0.48	0.47	0.47	0.49	0.33	0.32	0.33	0.32
5.0	0.44	0.42	0.45	0.45	0.43	0.45	0.43	0.42
6.0	0.40	0.42	0.40	0.45	0.51	0.51	0.51	0.49
7.0	0.37	0.33	0.35	0.43	0.57	0.62	0.57	0.56
8.0	0.33	0.30	0.31	0.33	0.64	0.65	0.63	0.62
9.0	0.27	0.28	0.28	0.29	0.69	0.67	0.68	0.67
10.0	0.25	0.25	0.26	0.23	0.71	0.70	0.70	0.73
11.0	0.20	0.21	0.20	0.20	0.77	0.75	0.77	0.76
12.0	0.17	0.17	0.19	0.18	0.81	0.80	0.78	0.80
13.0	0.16	0.15	0.16	0.17	0.83	0.83	0.82	0.83
14.0	0.14	0.14	0.16	0.15	0.85	0.84	0.82	0.83

Table 6.5: Mean warranted literals ratio and mean blocked literals ratio for $V = \{20.25, 30, 35\}$ $C/V = \{1..14\}$ and $f = 0.0$.

of more direct conflicts between contradictory literals, and the number of warranted literals will decrease, as well as the number of cyclic dependencies between arguments. So, when the presence of more direct conflicts increases in contrast to the number of indirect conflicts obtained by cyclic dependencies, the computation time will decrease at the same time that the number of outputs per instance.

We also observe that as the number of variables V increases, the maximum mean number of outputs seems to decrease. A possible explanation could be that as we have more variables but we do not increase the maximum length of the clauses (ML) the probability of having cyclic dependencies between arguments decreases.

7 Conclusions and Future Work

As we have already introduced, the contributions of this thesis can be divided in two groups: one about the definition, design and implementation of an argumentation framework for RP-DeLP programs, and the other about solving hard combinatorial arithmetic problems using SAT, PB and ASP encodings.

As contributions in the first group, we have presented an argumentation framework for RP-DeLP programs with two different semantics: a multiple outputs semantics and a maximal ideal output semantics. In the first semantics, we compute all the possible sets of conclusions that are warranted from the program (each set of conclusions has valid arguments that support them and they are consistent together), but for each set of warranted conclusions we also compute a set of blocked conclusions (a set of conclusions that also have valid arguments supporting them but that generate conflicts with the warranted conclusions). The way in which the different clauses of the program are distributed among the strict level and the defeasible levels determine whether a conclusion with a valid argument is finally warranted or blocked. We have presented the design of an algorithm for computing the outputs of an RP-DeLP program, and we have implemented it. In a first version of the algorithm, possible conflicts between arguments are detected checking all the possible cycles in a warrant dependency graph. Then, in order to get a more efficient version for implementation, we presented an improved version that avoids the computation of the warrant dependency graph and checking cycles on it. In the improved version, the main algorithm does all its work of detecting warranted and blocked conclusions in outputs by solving two main subproblems: finding almost valid arguments and finding conflict sets of arguments.

Then, still with the goal of achieving an scalable algorithm implementation, we presented transformations of these two subproblems to SAT and ASP encodings, in order to get advantage of the current performance of state-of-the-art SAT and ASP solvers. The SAT encoding for finding almost valid arguments follows an analogous approach to the one used for solving classical STRIPS planning problems with SAT encodings, and the SAT encoding for finding conflict sets of arguments is actually an extension of the previous one: we now search for a set of arguments, still of just one, but with the additional constraint that together should generate a contradiction. The ASP encoding approach follows a similar idea, being the main difference that as ASP inherits its language from the language of logic programming (like RP-DeLP does), so facts and rules in ASP can be used directly for facts and rules on an RP-DeLP program. In addition, there is no need to explicitly encode the non-deterministic choice of clauses for inferring conclusions in an argument or to encode the different stages of evolution of the true literals of an argument with different sets of variables.

In the second semantics we consider only one set of warranted and blocked conclusions, called the maximal ideal output, that tries to capture the best possible consensus about a maximal set of warranted conclusions that are free from any possible conflicts. While in the first semantics circular definitions of conflicts were solved by producing a new output for each conclusion involved in the conflict, in the maximal ideal output this deadlock situation is solved by adding all the conclusions into the set of blocked literals. So, any conflict between valid arguments is resolved by blocking all of them and in the final set of warranted conclusions only conclusions derived from valid arguments that are free from circular and collective conflicts are included. So, there is no possible way to argue against any warranted conclusion in the maximal ideal output. Compared with the outputs obtained under the first semantics, the typical maximal ideal output will tend to have less warranted conclusions and more blocked conclusions. The algorithm designed, and implemented, for solving RP-DeLP programs with this second semantics is very similar to the first one, being the main difference that now it discovers the warranted and blocked literals following the recursive definition of warranted and blocked literals but with an iterative algorithm. That is, it first discovers simpler warranted and blocked literals before being able to find more complex ones. By contrast, in the multiple outputs algorithm, each time a circular conflict between arguments is found, it performs recursive calls to the main algorithm, to split the search of outputs, with one call for every literal involved in the circular conflict. But the main computational queries, looking for almost valid arguments and conflict sets of arguments, are solved in the same way. One of the most relevant properties of our algorithms is that although the possible set of arguments that can be build from an RP-DeLP program can be exponentially large, we do not need to maintain in memory at any time an exponentially large set of arguments, because in the worst case we need to maintain one argument for each conclusion, even when looking for conflict sets.

Then, with the goal of making our argumentation framework more available for a wider research community, we presented a web based version of our RP-DeLP framework, that allows to solve RP-DeLP programs using both semantics, and it not only computes and gives outputs, with warranted and blocked conclusions, but also informs the user about the arguments supporting warranted conclusions and the conflict sets found that explain the blocked conclusions. Having an available reasoner as a web application has many advantages, specially to be an easy-to-use tool for non-expert potential users.

Finally we performed an empirical study of the performance of the argumentation algorithms implemented using randomly generated problem instances. We developed a problem generator that allows to select the different parameters of an RP-DeLP program that can make a difference regarding the complexity of solving it: the number of variables, the number of defeasible levels, the amount of clauses in each level, and the maximum length of the clauses. The results indicate that a first important parameter regarding the complexity of solving the programs is the amount of strict knowledge. Only when this quantity is very small we get programs where many possible valid arguments can be obtained, so the complexity of deciding which literals are warranted or blocked can be high. But when the amount of strict knowledge exceeds a certain value, almost any possible argument is not valid, given the closure and consistency properties that need to be satisfied with respect to the strict knowledge. When the strict knowledge is small, or there is not strict knowledge at all, the parameters that affect the complexity are the ratio of clauses to variables, and how the clauses are distributed among the different defeasible levels. A interesting result is that the complexity seems to be lower when the clauses are more balanced between the different levels, because them the number of blocked literals generated in each level is decreased and the computation of blocked conclusions takes less time in each level, compared to what happens when in some level a high amount of clauses causes the generation of a high number of blocked conclusions. Regarding the difference between the SAT and the ASP based approaches, our results indicate that the ASP approach outperforms the SAT based. We presented preliminary explanation of why we think that ASP performs better.

We have seen that the ASP approach outperforms the SAT one. The reason for this better performans still remains to be more explained, but one reason may be that ASP represents information with rules and facts, just like RP-DeLP does.

As contributions related to using SAT, PB and ASP encodings for solving other hard combinatorial problems, we have considered several arithmetic problems and one problem about communication networks design. As arithmetic problems, we considered the problems of integer number factorization and discrete logarithm, and tried to design encodings with good propagation properties regarding the unit propagation technique used by SAT solvers. We also presented SAT and PB encodings for solving modular constraints. The results show that we can outperform other SAT encoding approaches to solve problems with such constraints. Finally, we presented SAT, ASP and PB encodings for solving the routing and wavelength assignment (RWA) problem, which is a problem that arises when a route must be established in an all-optical communication network. Our approach shows a better performance than other PB based previous approaches in instances where the size of the network is significant and the resources (wavelengths and paths) are critically constrained.

As future work, we present several lines. Regarding the semantics studied for our argumentation framework, although they are very appealing given that they satisfy the rationality postulates of Caminada and Amgoud, it would be interesting to consider other possible semantics that may be relevant in certain application domains. Regarding worst-case complexity, we lack an study of the worst-case complexity of our problems, and we plan to study this considering reductions from other well studied argumentation and non-monotone reasoning problems where worst-case results are known. With respect to the performance obtained

As a possible way to improve our algorithms, an interesting direction would be to present SAT and ASP encodings able to find minimal conflict sets, given that the current encodings do not ensure that the conflict sets found are minimal, and minimal conflict sets can be more informative for the final user, although its computation will tend to be more complex. Finally, one of the most intriguing results we have obtained is the superiority of the ASP based approach with respect to the SAT based one. Although we presented possible reasons for this performance difference, based mainly on the fact of the more close nature of RP-DeLP to ASP than to SAT so we retain more *structure* of the original problem with ASP, it would be interesting to deep into the reasons behind this performance difference, given that in previous SAT solvers competitions when ASP solvers are used they tend to be only better on unsatisfiable instances, but in our problem instances considered in the experimental evaluation the instances that show a clear advantage of ASP with respect to SAT are always feasible instances, so the reasons of these superior performance of ASP should be investigated with more detail.

Bibliography

[ABF$^+$10] Carlos Ansótegui, Ramón Béjar, Cèsar Fernández, Carla Gomes, and Carles Mateu. Generating highly balanced sudoku problems as hard problems. *Journal of Heuristics*, pages 1–26, 2010. 10.1007/s10732-010-9146-y.

[ABG10] Teresa Alsinet, Ramón Béjar, and Lluis Godo. A characterization of collective conflict for defeasible argumentation. In *Computational Models of Argument: Proceedings of COMMA 2010*, volume 216 of *Frontiers in Artificial Intelligence and Applications*, pages 27–38. IOS Press, 2010.

[ACGS08] Teresa Alsinet, Carlos I. Chesñevar, Lluis Godo, and Guillermo R. Simari. A logic programming framework for possibilistic argumentation: Formalization and logical properties. *Fuzzy Sets and Systems*, 159(10):1208–1228, 2008.

[AGKS00] Dimitris Achlioptas, Carla Gomes, Henry Kautz, and Bart Selman. Generating satisfiable problem instances. In *Proceedings of the AAAI 2000*, pages 256–261. AAAI Press / The MIT Press, 2000.

[Amg12] Leila Amgoud. Postulates for logic-based argumentation systems. In *Proceedings of the ECAI-2012 Workshop WL4AI*, pages 59–67, 2012.

[ANORC09] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In *SAT'09*, pages 167–180, 2009.

[AP04] Dimitris Achlioptas and Yuval Peres. The threshold for random k-sat is $2^k \log 2 - \mathcal{O}(k)$. *Journal of the American Mathematical Society*, 17(4):947–973, 2004.

[ARA07] Fadi A. Aloul, Bashar Al Rawi, and Mokhtar Aboelaze. Routing in optical and non-optical networks using boolean satisfiability. *Journal of Communications*, 2(4):49–56, 2007.

[Bar03] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2003.

[Bat68] K.E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Comput. Conf.*, pages 307–314, 1968.

[BB03] Olivier Bailleux and Yacine Boufkhad. Efficient cnf encoding of boolean cardinality constraints. In *CP*, pages 108–122, 2003.

[BBR06] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. A translation of pseudo boolean constraints to sat. *JSAT*, 2(1-4), 2006.

[BBR09] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New encodings of pseudo-boolean constraints into cnf. In *SAT*, pages 181–194, 2009.

[BDKT97] Andrei Bondarenko, Phan Minh Dung, Robert A. Kowalski, and Francesca Toni. An abstract, argumentation-theoretic approach to default reasoning. *Artif. Intell.*, 93:63–101, 1997.

[BH00] Philippe Besnard and Anthony Hunter. Towards a logic-based theory of argumentation. In *AAAI/IAAI*, pages 411–416, 2000.

[BH08] Philippe Besnard and Anthony Hunter. *Elements of Argumentation*. The MIT Press, 2008.

[Bie] Armin Biere. Precosat version 236. http://fmv.jku.at/precosat.

[Boo51] Andrew D. Booth. Signed binary multiplication technique. *Q J Mechanics Appl Math*, 4(2):236–240, 1951.

[CA07] Martin Caminada and Leila Amgoud. On the evaluation of argumentation formalisms. *Artif. Intell.*, 171(5-6):286–310, 2007.

[CG93] Mats Carlsson and Mats Grindal. Automatic frequency assignment for cellular telephones using constraint satisfaction techniques. In *Int. Conf. on Logic Programming, (ICLP'93)*, pages 647–665, 1993.

[CGK92] I. Chlamtac, A. Ganz, and G. Karmi. Lightpath communications: an approach to high bandwidth optical wan's. *Communications, IEEE Transactions on*, 40(7):1171 –1182, July 1992.

[CIR$^+$11] F. Calimeri, G. Ianni, F. Ricca, M. Alviano, A. Bria, G. Catalano, S. Cozza, W. Faber, O. Febbraro, N. Leone, M. Manna, A. Martello, C. Panetta, S. Perri, K. Reale, M. Santoro, M. Sirianni, G. Terracina, and P. Veltri. The third answer set programming competition: preliminary report of the system competition track. In *Proceedings of the 11th international conference on Logic programming and nonmonotonic reasoning*, LPNMR'11, pages 388–403, Berlin, Heidelberg, 2011. Springer-Verlag.

[CML00] Carlos I. Chesñevar, Ana G. Maguitman, and Ronald P. Loui. Logical Models of Argument. *ACM Computing Surveys*, 32(4):337–383, December 2000.

[CSAG04] Carlos Iván Chesñevar, Guillermo Ricardo Simari, Teresa Alsinet, and Lluis Godo. A logic programming framework for possibilistic argumentation with vague knowledge. In *UAI*, pages 76–84, 2004.

[Dad68] L. Dadda. Some schemes for parallel multipliers. *Alta Frequenza*, 2:14–17, 1968.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[DMT06] Phan Minh Dung, Paolo Mancarella, and Francesca Toni. A dialectic procedure for sceptical, assumption-based argumentation. In *Computational Models of Argument: Proceedings of COMMA 2008*, volume 172 of *Frontiers in Artificial Intelligence and Applications*, pages 145–156. IOS Press, 2006.

[DMT07] Phan Minh Dung, Paolo Mancarella, and Francesca Toni. Computing ideal sceptical argumentation. *Artif. Intell.*, 171(10-15):642–674, 2007.

[Dun95a] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77(2):321 – 357, 1995.

[Dun95b] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–358, 1995.

[DVB$^+$09] Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Miroslaw Truszczynski. The second answer set programming competition. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 5753 of *Lecture Notes in Computer Science*, pages 637–654. Springer Berlin / Heidelberg, 2009.

[DW11a] Christian Drescher and Toby Walsh. Translation-based constraint answer set solving. In *IJCAI*, pages 2596–2601, 2011.

[DW11b] Christian Drescher and Toby Walsh. Translation-based constraint answer set solving. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 2596–2601, 2011.

[ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *JSAT*, 2(1-4):1–26, 2006.

[FMM03] Claudia Fiorini, Enrico Martinelli, and Fabio Massacci. How to fake an rsa signature by encoding modular root finding as a sat problem. *DISCRETE APPL. MATH*, 130:101–127, 2003.

[GDS09] Alejandro J. García, Jürgen Dix, and Guillermo R. Simari. Argument-based logic programming. In Iyad Rahwan and Guillermo R. Simari, editors, *Argumentation in Artificial Intelligence*, chapter 8, pages 153–171. Springer, 2009.

[GFD09] Patrick Gallagher, Deputy Director Foreword, and Cita Furlani Director. Fips pub 186-3 federal information processing standards publication digital signature standard (dss), 2009.

[GKK$^+$08] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A users guide to gringo, clasp, clingo, and iclingo, 2008.

[GKK$^+$11a] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):105–124, 2011.

[GKK$^+$11b] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):105–124, 2011.

[GKNS07] Martin Gebser, Benjamin Kaufmann, Andre Neumann, and Torsten Schaub. clasp: A conflict-driven answer set solver. In *In LPNMR'07*, pages 260–265. Springer, 2007.

[GLN$^+$07] Martin Gebser, Lengning Liu, Gayathri Namasivayam, André Neumann, Torsten Schaub, and Mirosł aw Truszczyński. The first answer set programming system competition. In *Proceedings of the 9th international conference on Logic programming and nonmonotonic reasoning*, LPNMR07, pages 3–17, Berlin, Heidelberg, 2007. Springer-Verlag.

[GS04a] A. García and G. Simari. Defeasible Logic Programming: An Argumentative Approach. *Theory and Practice of Logic Programming*, 4(1):95–138, 2004.

[GS04b] Alejandro J. García and Guilermo R. Simari. Defeasible Logic Programming: An Argumentative Approach. *Theory and Practice of Logic Programming*, 4(1):95–138, 2004.

[GvHSS07] Carla Gomes, Willem van Hoeve, Ashish Sabharwal, and Bart Selman. Counting csp solutions using generalized xor constraints. In *Proceedings of the 22nd National Conference on Artificial Intelligence*, 2007.

[GW97] Ian P. Gent and Toby Walsh. From approximate to optimal solutions: Constructing pruning and propagation rules. In *IJCAI'97*, pages 1396–1401, 1997.

[KMS96] Henry A. Kautz, David A. McAllester, and Bart Selman. Encoding plans in propositional logic. In *KR'96*, pages 374–384, 1996.

[KO04] Jeffery L. Kennington and Eli V. Olinick. Wavelength translation in wdm networks: Optimization models and solution procedures. *INFORMS Journal on Computing*, 16(2):174 –187, 2004.

[KS99a] Henry A. Kautz and Bart Selman. Unifying sat-based and graph-based planning. In *IJCAI'99*, pages 318–325, 1999.

[KS99b] Henry A. Kautz and Bart Selman. Unifying sat-based and graph-based planning. In *IJCAI*, pages 318–325, 1999.

[Lan00] H. W. Lang. Sequential and parallel sorting algorithms. In *http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/algoen.htm*, 2000.

[LPF$^+$06] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7(3):499–562, July 2006.

[MMS05] V.M. Manquinho and J. Marques-Silva. Effective lower bounding techniques for pseudo-boolean optimization [eda applications]. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 660–665 Vol. 2, March 2005.

[MMWW02] Hugues Marchand, Alexander Martin, Robert Weismantel, and Laurence A. Wolsey. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics*, 123(1-3):397–446, 2002.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography.* CRC Press, Inc., Boca Raton, FL, USA, 1996.

[NO06] Roberto Nieuwenhuis and Albert Oliveras. On sat modulo theories and optimization problems. In *SAT'06*, pages 156–169, 2006.

[Pol78] J.M. Pollard. Monte carlo methods for index computation mod p. *Mathematics of Computation*, 32:918–924, 1978.

[Pol09] John L. Pollock. A recursive semantics for defeasible reasoning. In Iyad Rahwan and Guillermo R. Simari, editors, *Argumentation in Artificial Intelligence*, chapter 9, pages 173–198. Springer, 2009.

[Pom85] C Pomerance. The quadratic sieve factoring algorithm. In *Proc. of the EUROCRYPT 84 workshop on Advances in cryptology: theory and application of cryptographic techniques*, pages 169–182, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[Pro96] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *AI Journal*, 81:81–109, 1996.

[PS97] Henry Prakken and Giovanni Sartor. Argument-based extended logic programming with defeasible priorities. *Journal of Applied Non-classical Logics*, 7:25–75, 1997.

[PV02] Henry Prakken and Gerard Vreeswijk. Logical Systems for Defeasible Argumentation. In D. Gabbay and F.Guenther, editors, *Handbook of Phil. Logic*, pages 219–318. Kluwer, 2002.

[RM09] Olivier Roussel and Vasco M. Manquinho. Pseudo-boolean and cardinality constraints. In *Handbook of Satisfiability*, pages 695–733. 2009.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[Sim09] Helmut Simonis. A hybrid constraint model for the routing and wavelength assignment problem. In *Proceedings of the 15th international conference on Principles and practice of constraint programming*, CP'09, pages 104–118, Berlin, Heidelberg, 2009. Springer-Verlag.

[Sin05] Carsten Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. In *CP*, pages 827–831, 2005.

[SK96] B. Selman and S. Kirkpatrick. Critical behaviour in the computational cost of satisfiability testing. *Artificial Intelligence Journal*, 81:273–295, 1996.

[SKC94] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *In Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-94*, pages 337–343, 1994.

[SLM92] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI'92, pages 440–446. AAAI Press, 1992.

[SNS02] Patrik Simons, Ilkka Niemelá, and Timo Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, June 2002.

[TK08] P.N. Tran and U. Killat. An exact ILP formulation for optimal wavelength converter usage and placement in WDM networks. In *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, pages 1 –6, December 2008.

[TKI08] Matthias Thimm and Gabriele Kern-Isberner. On the relationship of defeasible argumentation and answer set programming. In *Proceedings of Computational Models of Argument, (COMMA 2008)*, pages 393–404, 2008.

[TS11] Francesca Toni and Marek Sergot. Argumentation and answer set programming. In Marcello Balduccini and Tran Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, volume 6565 of *Lecture Notes in Computer Science*, pages 164–180. Springer Berlin / Heidelberg, 2011.

[Tse68] G. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Cosntr. Math. and Math. Logic*, 1968.

[VSK05] J. Valavi, N. Saluja, and S.P. Khatri. A boolean satisfiability based solution to the routing and wavelength assignment problem in optical telecommunication networks. In *Communications, 2005. ICC 2005. 2005 IEEE International Conference on*, volume 3, pages 1802 – 1806 Vol. 3, May 2005.

[Wax88] B.M. Waxman. Routing of multipoint connections. *Selected Areas in Communications, IEEE Journal on*, 6(9):1617 –1622, December 1988.

[XL00] K. Xu and W. Li. Exact phase transition in random constraint satisfaction problems. *Journal of Artificial Intelligence Research, (JAIR)*, 12:93–103, 2000.

[ZTTD03] Yongbing Zhang, Koji Taira, Hideaki Takagi, and Sajal K. Das. Adaptative wavelength routing and assignment in optical wdm networks. *Optical Networks Magazine*, 3(5):86–99, 2003.