Answer Set Programming: Founded Bounds and Model Counting

Rehan Abdul Aziz

Submitted in total fulfilment of the requirements of the degree of
Doctor of Philosophy

Department of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

September 2015
Abstract

A

NSWER Set Programming (ASP) is a powerful modelling formalism that is very efficient for solving combinatorial problems. This work extends the state-of-the-art in theory and practice of ASP. It has two parts.

ASP solvers implement the stable model semantics that eliminates circular derivations between Boolean variables from the solutions of a logic program. Due to this, ASP solvers are better suited than propositional satisfiability (SAT) and Constraint Programming (CP) solvers for solving a certain class of problems whose specification includes inductive definitions such as reachability in a graph. On the other hand, ASP solvers suffer from the grounding bottleneck that occurs due to their inability to model finite domain variables. Constraint Answer Set Programming (CASP) is one approach that combines the strengths of CP and ASP to remove the grounding bottleneck. We present a CASP system that extends a CP solver. This is in contrast with other CASP systems that embed CP-type reasoning on finite bounds inside an ASP solver.

CASP systems cannot remove another type of bottleneck that arises due the inability of a system to prevent circular derivation on numerical bounds of variables. An example where this sort of reasoning is required is in modelling shortest paths between nodes in a graph. Just as reachability can be encoded as an inductive definition with one or more base cases and recursive rules, shortest paths between nodes can also be modelled with similar base cases and recursive rules for their upper bounds. We show that the existing stable model semantics is insufficient to disallow circular reasoning on the bounds of numeric variables. Due to this deficiency, this type of grounding bottleneck cannot be removed by CASP systems, which naively merges stable model semantics on ASP variables with classical reasoning of CP over numerical variables. For this purpose, we re-
quire a theoretical extension of the semantics from Booleans and normal rules to bounds over numeric variables and more general rules. We propose Bound Founded Answer Set Programming (BFASP) that resolves this issue and consequently, removes both types of grounding bottleneck mentioned above.

The second part of this work deals with stable model counting. Counting the models of a given propositional theory, or simply, model counting, is a well studied problem. Many modern model counters are based on DPLL-based search which is commonly used in SAT solvers. Due to this common ground, they borrow many useful features like nogood learning from SAT solvers. At the same time, the best ASP solvers are extensions of SAT solvers. This motivates us to implement a stable model counter by taking an existing propositional model counter and adding the additional elements that turn a SAT solver into an ASP solver. We solve this problem from a theoretical as well as a practical perspective.

Stable model counting has application in solving inference tasks in probabilistic logic programming. In probabilistic logic programming, a problem is modelled as a logic program where the random variables are the decision variables and their relations are specified as normal logic rules, possibly by introducing some definitional or derived atoms. The observed evidence is simply a set of unary constraints on the decision variables and derived atoms. The goal of the inference task is to compute the probability of certain query atoms being true. Due to close relation of the distribution semantics of a probabilistic logic program and the stable model semantics of a logic program, the inference task is essentially reduced to stable model counting. In some state-of-the-art probabilistic logic programming systems, stable model counting is done by translating the logic program to a SAT program and using a propositional model counter. We show how we can use our stable model counting algorithms to solve the inference tasks in probabilistic logic programming and compare it with existing translational approaches.

Finally, we look at the task of projected stable model counting. In this problem, the goal is to compute the number of stable models on a projected vocabulary. Again, we approach the problem through SAT. Surprisingly, there has been very little work on projected model counting. Therefore, our major contribution is in proposing several algo-
rithms for projected model counting. We then extend these algorithms to perform projected stable model counting.

This thesis has the following contributions:

- An implementation of Constraint Answer Set Programming within a lazy clause generation constraint solver.

- The theory of Bound Founded Answer Set Programming (BFASP), a formalism that generalizes ASP and CP.

- An algorithm for implementing BFASP within a lazy clause generation constraint solver. The algorithm allows direct reasoning with founded numeric variables and complex rule forms, and as a result, is able to solve some benchmarks more efficiently than competing solvers.

- An extension of the following language techniques for BFASP: flattening, bottom-up grounding, and magic set transformation. Flattening is used in CP, while the latter two come from logic programming.

- A novel algorithm for counting stable models of a program which is implemented in a propositional model counter. The model counter that we use for our implementation is actually a knowledge compiler, built on top of a model counter. Our addition of stable model counting capabilities to the propositional model counter naturally enables the solver to perform stable model knowledge compilation.

- Implementation and comparison of three algorithms for projected model counting; the first algorithm is not novel, the second extends an existing algorithm and the third one is a novel algorithm based on knowledge compilation. We describe how all three algorithms can be adapted for projected stable model counting.
Declaration

This is to certify that

1. the thesis comprises only my original work towards the PhD,

2. due acknowledgement has been made in the text to all other material used,

3. the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Rehan Abdul Aziz, September 2015
Acknowledgements

I am deeply grateful to my supervisors Peter Stuckey and Geoffrey Chu for their guidance throughout my candidature. Peter has been the perfect supervisor; exceptionally smart, extremely knowledgeable, genuinely passionate about research, possessing flawless work ethic, and above all, sincerely believing in his students and the work they do. Any PhD student can attest that supervisors that are remarkable in all these qualities are rare. I really appreciate Peter’s work ethic regarding regular weekly meetings with students despite such a busy schedule. I surely would have drifted to a deadlock without these regular meetings. My co-supervisor Geoffrey has also been extremely generous with his time. Working with him and learning from his exceptional problem solving ability has been profoundly enlightening. His insightful advise on everything, from theory to software, has significantly deepened my understanding and helped me a great deal in achieving my PhD goals. Peter and Geoffrey are a source of inspiration for me and what I have learned from them will stay with me for my remaining career.

I would also like to thank Zoltan Somogyi who was my co-supervisor during the first year. His input was central in providing me with a good start, which is crucial in PhD. I also want to thank my co-author Christian Muise, who regularly joined our weekly meetings in the last year, and contributed greatly to our discussions and ideas. He helped me out promptly whenever I needed to discuss anything about theory or software, and for that, I am grateful. Thanks to my advisory committee member Adrian Pearce for gladly chairing every progress review session, and for providing very encouraging and useful feedback. Thanks to Guido Tack for helping me out with the MINIZINC library. Big thanks to everyone I worked with, it has been a very exciting and stimulating time and there has never been a pause in original ideas.
I am thankful to National ICT Australia and The University of Melbourne for funding my PhD, and the former for sponsoring my travel to conferences. Thanks to the anonymous reviewers of papers published during my PhD, and thanks to Martin Gebser and Jael Kriener for organizing the ICLP Doctoral Consortium 2014 that provided me with the opportunity to present my work to the specialist audience and get helpful feedback.

I greatly appreciate the friendly atmosphere in the department and I want to thank all the people responsible for creating and sustaining it; including but not limited to my fellow graduate students, the head of department and the administrative staff. Special thanks to all my friends, particularly the ones on the 6th floor, for being part of such a vibrant work-place. My years at the department would have been unbearably dull and inorganic without them. I will remember all the time spent at the water cooler, coffee breaks, lunches, and pizza talks discussing everything from science, to politics, to PhD woes.

I would like to thank my family and my in-laws for their love and support. Special thanks to my parents who always prioritized education over all other practical affairs, this PhD wouldn’t have been possible without that value. Lastly, a huge thanks to my loving wife Naurin for her ceaseless and warm support. She shared all pleasurable and painful moments of this journey with me, that we began and concluded together, as an enduring team. This thesis is dedicated to her.

Rehan Abdul Aziz,

September 2015.
Preface

This research was conducted at the Computing and Information Systems Department, University of Melbourne, from March 2012 to September 2015, in collaboration with my supervisors and co-authors Peter Stuckey, Geoffrey Chu and Zoltan Somogyi (who was my supervisor during the first year) and with co-author Christian Muise. Parts of this thesis have appeared in the following publications:


- Rehan Abdul Aziz, Geoffrey Chu, Christian J. Muise, and Peter James Stuckey. Stable model counting and its application in probabilistic logic programming. In Pro-

To Naurin
# Contents

1 Introduction .......................... 1
   1.1 Part I ........................................... 1
   1.2 Part II ......................................... 5
   1.3 Outline Of Chapters ..................... 6

2 Formal Background ...................... 9
   2.1 Decision Variables And Domains ............ 9
   2.2 Constraint Programming ....................... 10
   2.3 Boolean Satisfiability Problem (SAT) ......... 11
   2.4 Answer Set Programming (ASP) ................ 12
      2.4.1 Constraints And Standard Variables ....... 14
      2.4.2 Dependency Graph ....................... 16
      2.4.3 Clark Completion ....................... 16
      2.4.4 Tight Logic Programs ................... 18
      2.4.5 Unfounded Sets ....................... 19
   2.5 Aggregates In ASP ......................... 21
      2.5.1 Cardinality And Weight Constraint Rules .... 21
      2.5.2 Minimum And Maximum Aggregates .......... 23
   2.6 Fuzzy Answer Set Programming .............. 24
   2.7 Propositional Model Counting ............... 26
   2.8 Knowledge Compilation ..................... 27

3 Solving Algorithms .................... 29
   3.1 Finite Domain Constraint Solving .......... 29
   3.2 SAT Solving .................................. 33
      3.2.1 DPLL And Unit Propagation .............. 33
      3.2.2 Watched Literals ....................... 34
      3.2.3 Conflict Analysis, Nogood Learning And Backjumping .... 35
   3.3 Lazy Clause Generation ...................... 37
   3.4 Unfounded Set Propagation .................. 41
      3.4.1 Source Pointer Technique ............... 42
      3.4.2 ASP Via SAT + Loop Elimination .......... 43
      3.4.3 Unfounded Set Algorithms For Conflict-Driven Solvers .... 45
   3.5 Solving Weight Constraint Rules In ASP .. 45
   3.6 Model Counting ................................ 46
6.8 Related Work ...................................................... 107

7 BFASP Implementation .............................................. 109
  7.1 Strongest Bound And Stabilizers ............................... 109
  7.2 Overview And Analysis Of The Algorithm ..................... 111
    7.2.1 Justified Bounds ...................................... 111
    7.2.2 Unfounded Sets And Correctness Results ................. 113
  7.3 Algorithm .................................................... 115
  7.4 Experimental Results ....................................... 119

8 Language of BFASP ................................................ 123
  8.1 Non-Ground BFASPs ........................................... 123
  8.2 Flattening .................................................... 125
  8.3 Grounding ..................................................... 131
  8.4 Magic Set Transformation .................................... 135
    8.4.1 Background ............................................ 136
    8.4.2 Magic Set Transformation For BFASP .................... 140
    8.4.3 Unstratified BFASPs ................................... 143
  8.5 Experiments .................................................. 145
  8.6 Conclusion ................................................... 148

II Stable Model Counting ............................................. 149

9 Stable Model Counting and Probabilistic Logic Programming .... 151
  9.1 Introduction ................................................ 151
  9.2 SAT-Based Stable Model Counting ............................. 153
    9.2.1 Searching On Standard Variables For Stratified Programs 154
    9.2.2 Modifying The Residual Program .......................... 155
  9.3 PROBLOG2 Via Stable Model Counting ......................... 162
  9.4 Experiments .................................................. 164
  9.5 Conclusion ................................................... 167

10 Projected Stable Model Counting .................................. 169
  10.1 Introduction ............................................... 169
  10.2 Projected Model Counting .................................... 171
    10.2.1 Restricting Search To Priority Variables ............... 171
    10.2.2 Blocking Seen Solutions ................................ 173
    10.2.3 Counting Models Of Projected d-DNNF ................... 176
  10.3 Experiments ................................................ 180
    10.3.1 Uniform Random 3-SAT and Boolean Circuits .......... 181
    10.3.2 Planning ................................................. 186
  10.4 Projected Stable Model Counting .............................. 186
    10.4.1 DSHARP,P .............................................. 187
    10.4.2 #CLASP ................................................. 187
    10.4.3 D2C .................................................... 189
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Trace of DPLL algorithm</td>
<td>34</td>
</tr>
<tr>
<td>3.2</td>
<td>Implication graph for decisions $a$, $\neg d$, $\neg f$</td>
<td>36</td>
</tr>
<tr>
<td>3.3</td>
<td>Implication graph for a constraint program inside a lazy clause generation solver (source: Stuckey (2014))</td>
<td>40</td>
</tr>
<tr>
<td>3.4</td>
<td>Implication graph for a constraint program inside a lazy clause generation solver after learning nogood (source: Stuckey (2014))</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>ASP encoding of Minimum Dominating Set (MDS)</td>
<td>54</td>
</tr>
<tr>
<td>4.2</td>
<td>ASP encoding of Minimum Connected Dominating Set (MCDS)</td>
<td>55</td>
</tr>
<tr>
<td>4.3</td>
<td>ASP encoding of uniform width constraints for MCDS with uniform width (MCDS-UW)</td>
<td>59</td>
</tr>
<tr>
<td>4.4</td>
<td>ASP encoding of diameter constraint in MCDS with bounded diameter (MCDS-BD)</td>
<td>62</td>
</tr>
<tr>
<td>5.1</td>
<td>Initializing source pointers</td>
<td>72</td>
</tr>
<tr>
<td>5.2</td>
<td>Event notification for ANGER</td>
<td>74</td>
</tr>
<tr>
<td>5.3</td>
<td>Propagation for ANGER</td>
<td>74</td>
</tr>
<tr>
<td>5.4</td>
<td>Unfounded set algorithm for ANGER: (source Anger et al. (2006))</td>
<td>77</td>
</tr>
<tr>
<td>5.5</td>
<td>Propagation for GEBSER</td>
<td>80</td>
</tr>
<tr>
<td>5.6</td>
<td>Unfounded set algorithm for GEBSER (source: Gebser et al. (2012b))</td>
<td>81</td>
</tr>
<tr>
<td>6.1</td>
<td>BFASP encoding for the shortest path problem</td>
<td>103</td>
</tr>
<tr>
<td>6.2</td>
<td>A graph of roads and distances</td>
<td>103</td>
</tr>
<tr>
<td>6.3</td>
<td>BFASP encoding for the road construction problem</td>
<td>104</td>
</tr>
<tr>
<td>6.4</td>
<td>BFASP encoding of Utilitarian Policies</td>
<td>105</td>
</tr>
<tr>
<td>6.5</td>
<td>BFASP encoding of Company Controls</td>
<td>106</td>
</tr>
<tr>
<td>6.6</td>
<td>Translation of Fuzzy ASP to BFASP</td>
<td>107</td>
</tr>
<tr>
<td>7.1</td>
<td>Main propagation algorithm</td>
<td>117</td>
</tr>
<tr>
<td>7.2</td>
<td>Processing domain change events</td>
<td>118</td>
</tr>
<tr>
<td>7.3</td>
<td>Main unfounded set calculation</td>
<td>119</td>
</tr>
<tr>
<td>7.4</td>
<td>Explaining unfounded sets</td>
<td>120</td>
</tr>
<tr>
<td>8.1</td>
<td>Creating constraint programs for grounding rules</td>
<td>134</td>
</tr>
<tr>
<td>8.2</td>
<td>Pseudo-code for bottom-up grounding</td>
<td>135</td>
</tr>
<tr>
<td>9.1</td>
<td>Execution of PROBLOG2</td>
<td>163</td>
</tr>
</tbody>
</table>
9.2 SmokersFriends with 31 random variables ........................................ 166

10.1 A visualization of the search tree for model counting with priority vari-
ables. Nodes are marked with residual clauses and counts. Dotted edges
indicate dynamic decomposition, dashed edged indicate backjumps over
non-priority decisions. ................................................................. 173

10.2 Pseudo-code for shrinking a solution $\theta$ of original clauses $C$ and blocking
clauses $B$ to a solution cube $S$, adding its count and a blocking clause to
prevent its reoccurrence. ......................................................... 175

10.3 A visualization of counting models via blocking solutions. The curly arcs
indicate free search, ending in a solution, with an associated count. The
controlled search is indicated by full arcs, and blocking clauses associated
with controlled search decisions are shown on arcs. ........................ 176

10.4 Pseudo-code for projected model counting via counting models of CNF
encoding of projected DNNF ....................................................... 179

10.5 Example of application of d2c ................................................. 180

10.6 Pseudo-code for shrinking a solution $\theta$ of an original ASP program $P =$
$(V, R, C)$ and blocking clauses $B$ to a solution cube $S$, adding its count and
a blocking clause to prevent its reoccurrence. .............................. 188
# List of Tables

1.1 Features in different formalisms ............................................. 2
3.1 Constraint propagation leading to a failure (source: Stuckey (2014)) .... 32
3.2 Backtracking after a failure (source: Stuckey (2014)) .................... 32
5.1 Results for routing on 34 instances ....................................... 83
5.2 ASP problems, geometric restart, no timeout ............................... 85
5.3 ASP problems, Luby restart, with timeout .................................. 86
7.1 ShortPath ............................................................... 122
7.2 RoadCon ............................................................... 122
7.3 CompCon ............................................................... 122
7.4 UtilPol comparison, and scaling behaviour on the smallest instance ..... 122
8.1 Grounding conditions for rule $r = (c, y)$ .................................. 132
8.2 Road Construction RoadCon .............................................. 146
8.3 UtilPol ............................................................... 146
8.4 CompanyCon ............................................................ 147
9.1 Comparison of PROBLOG2, ASPROBLOG, and ASPROBLOGS on the Graph Reliability problem ................................................. 165
10.1 Results from random uniform 3-SAT and Boolean circuits ............... 185
10.2 Results from robustness of partially ordered plans to initial conditions .. 187
Chapter 1
Introduction

A NSWER Set Programming (ASP) (Baral, 2003, Gebser et al., 2012a, Gelfond and Lifschitz, 1988b, Marek and Truszcynski, 1998, Niemelä, 1999) is a declarative programming paradigm based on stable model semantics for logic programs. Broadly, this dissertation contributes to the theory and practice of ASP. More particularly, the thesis contributes to two areas in ASP. The first contribution deals with combining the strengths of ASP systems with Constraint Programming (CP) (Marriott and Stuckey, 1998, Rossi et al., 2006) systems as well as extending those strengths from Boolean to numeric variables, from which CP systems can benefit. The second one deals with stable model counting. Let us now briefly discuss both contributions in more detail.

1.1 Part I

In the first contribution of the thesis, we address some shortcomings of ASP and CP systems. Modern ASP and CP systems are geared towards solving combinatorial problems. The focus of this part of thesis is on certain features of these solvers. The first useful feature is the ability to represent and reason with finite domain and real variables. The second useful feature is foundedness, which we discuss in great detail in the later chapters of this thesis. Intuitively, foundedness is a property associated with certain variables in a formalism that captures the idea that the value that a variable takes must be justified (founded) by one or more rules, and this foundedness among multiple variables cannot be circular. In the absence of any rule, the value of the variable defaults to a certain value, i.e., a certain uninteresting or trivial value is founded by default. A widely used example
is that of reachability in a graph, where the propositional variables represent whether a node is reachable from some other node. The default value is false, and the truth of reachability from $a$ to $b$ needs to be justified by either a direct edge from $a$ to $b$, or the fact that $a$ can reach another node $c$ that has an edge to $b$.

Table 1.1 provides an overview of the motivation and subject matter of Part I of this thesis. The table lists formalisms as rows, and has certain features as columns. The features include the types of variables, supported constraints, variables that can be founded, and valid rules in the formalism. The systems that we consider are propositional or Boolean satisfiability (SAT), ASP, CP, a combination of ASP and CP called Constraint ASP (CASP) (Gebser et al., 2009c), a system that we propose called Bound Founded ASP (BFASP), and Fuzzy ASP (FASP) (Blondeel et al., 2013, Nieuwenborgh et al., 2006).

There are some caveats that must be noted about the table. First of all, it is only intended as a guide, or a broad map, to acquaint the reader with the direction that we take in subsequent chapters. We have attempted to list the formalisms in their most widely known formats, but this does not mean that there are not extensions of a formalism, and algorithms for those extensions, that support a broader feature set. For example, pseudo-
Boolean constraints are also used in SAT, but they are not as common as they are in ASP systems; for which the semantics of weight constraints as well as specialized algorithms for dealing with them is an essential feature. Secondly, the feature list is not exhaustive and there are other important features, such as implementation of languages, support for optimization, efficient algorithms etc that are not included for conciseness. Thirdly, for CP, CASP, and BFASP, a given implementation might only support a subset of features of the formalism. For ASP and SAT implementations, this gap between theory and practice is mostly non-existent, while for FASP, there is no well-known implementation. The gap in most CP solvers arises due to lack of support for floats, which arises due to practical reasons of fixed point-precision, although there is work on algorithms for CP solving that use interval reasoning (Benhamou, 1994) on continuous domain variables. The important point is that all CASP and BFASP implementations that we discuss in the thesis, that use or extend a CP solver that does not support continuous variables, inherit this limitation. Let us now discuss each formalism.

The first entry is SAT which in its most widely known form only supports Boolean variables and a theory in Conjunctive Normal Form (CNF) (although practically all Boolean formulas can be efficiently transformed into CNF) and does not support foundedness at all. The next entry is ASP. Traditionally in the ASP literature, the variables are called atoms but we simply refer to them as Boolean variables. The types of constraints that an answer set program can have are integrity constraints (same as SAT clauses) and most modern ASP systems also support Pseudo-Booleans (a Pseudo-Boolean can be written as a weight constraint in ASP). One of the unique strengths of ASP is foundedness over the Boolean variables. Finally, the rule forms that it allows are normal logic rules, as well as weight constraint rules which we describe in detail when we review the preliminaries.¹

For CP, the supported variables in theory include both finite domain (e.g. integer, Boolean, sets) as well as floats. Any constraint, intensional or extensional, in theory, is supported. The constraints are usually divided into primitive and global categories, although the division is rather arbitrary. There is no notion of foundedness or rules in

¹As mentioned by Simons et al. (2002) and Gebser et al. (2009a), weight constraint rules can linearly capture normal rules and cardinality rules/constraints. They can also replace the other two used rule aggregates: minimum and maximum, as we describe in Chapter 2.
Next, in CASP, each feature can be considered as simply the sum of features of ASP and CP. In particular, the variable set is the same as that of CP. Furthermore, it borrows foundedness over Booleans and rules from ASP. The formalism that we propose in this thesis, BFASP, generalizes all the systems above it and most importantly, its core novelty lies in adding foundedness over numeric variables. The set of rules is practically the same as the set of constraints, as long as certain requirements that we describe later, are satisfied. Fuzzy ASP also supports founded numeric variables and rule forms based on certain predefined norms, and we shall say more on how it relates to BFASP in the later chapters.

BFASP is a significant generalization of ASP and CASP. In particular, it supports founded numeric variables and allows for much more general rule forms. We briefly describe the intuition of founded numeric variables here in order to give the reader a feel of what BFASP allows us to do. Earlier, we looked at how in ASP, due to foundedness over Boolean variables, we can model reachability in a graph by describing a base case and an inductive case. BFASP extends this type of reasoning to numeric variable, and an example where this is useful is when we are interested in modelling shortest paths in a graph. Just like reachability variables which are false by default and the truth value needs to be founded by some rule, shortest path variables are numeric founded variables that are equal to infinity by default in absence of any rule, and further rules justify a lower value. The base case for shortest path variables says that the shortest path from any variable to itself is 0, and the inductive case specifies that for every edge \((u, v)\) and every node \(x\), the shortest path from \(x\) to \(v\) is at most the shortest path from \(x\) to \(u\) plus the weight of the edge \((u, v)\). Of course, shortest path, considered in isolation, is not a combinatorial problem, but there could be combinatorial problems that require access to shortest path variables, in order to constrain them, or include them in the objective function. Such problems can be modelled elegantly using the shortest path rules we just described and solved efficiently using BFASP algorithms. An example benchmark is when we want to build a road network subject to monetary constraints and our goal is to minimize the sum of pairwise shortest paths in the region. Our experiments suggest that using BFASP algorithms, we can solve such problems more efficiently than implementations of other
1.2 Part II

Model counting is the problem of computing the number of models that satisfy a given propositional theory. It is denoted as \( \#\text{SAT} \) and is a well-studied problem. In this part of the thesis, we look at the problem of counting answer sets of a given logic program which we denote it as \( \#\text{ASP} \). There are two chapters in this part, and each chapter is a contribution.

The first chapter describes techniques for solving \( \#\text{ASP} \). \( \#\text{ASP} \) has an important application in solving inference tasks in probabilistic logic programming, where the goal is to compute the probability of given queries being true provided a set of mutually independent random variables, a model (a logic program) that relates these variables possibly by introducing some definitions, and some evidence modelled as fixing the truth values of some variables in the model. So far, this has been done by translating the logic program to a propositional theory and using a \( \#\text{SAT} \) solver. The probability of a query atom \( q \) being true is equal to the ratio: \( \frac{\text{WMC}(T\land q)}{\text{WMC}(T)} \) where the numerator is the weighted model count of the translated theory and the query, and the denominator is the weighted model count of only the theory. The weight of a model is equal to the product of probabilities of truth values of all random variables in that model, and the weighted model count of a theory is the sum of weights of all models that satisfy the theory. In the chapter, we show that for some problems that involve inductive definitions like reachability in a graph, the translation of logic programs to SAT can be expensive for the purpose of solving inference tasks. For such problems, direct implementation of stable model semantics allows for more efficient solving. We present two implementation techniques, based on unfounded set detection, that extend a propositional model counter to a stable model counter.

In the second chapter of this part, we look at the problem of counting stable models of a logic program projected on a subset of variables that we call priority variables. The task is to compute the number of assignments to these variables such that there exists an extension to the rest of the variables, non-priority variables, which is an answer set of the
original problem. We denote this as \( \#\exists \text{ASP} \). In the chapter, we first describe how to solve \( \#\exists \text{SAT} \) and then extend that to ASP. We discuss three different approaches to \( \#\exists \text{SAT} \) (two of which are novel), and compare their performance on different benchmark problems.

1.3 Outline Of Chapters

Here is the chapter outline of the thesis:

- Chapter 2 provides the relevant formal background for the rest of the thesis. We look at definitions, semantics, and properties of Constraint Programming, Answer Set Programming, Model Counting and Knowledge Compilation that are relevant for understanding the subsequent chapters.

- Chapter 3 describes the implementation techniques and solving algorithms for the formal systems defined in Chapter 2.

- In Chapter 4, we provide the motivation for combining ASP and CP. In particular, we look at strengths and weaknesses of ASP and CP. We describe two types of grounding bottlenecks: standard and founded. The standard grounding bottleneck can be removed by combining the strengths of ASP and CP, and this is described in detail in Chapter 5. The founded grounding bottleneck cannot be removed by naively combining ASP and CP, instead, it requires a theoretical extension of stable model semantics used by ASP solvers. Parts of this chapter have appeared in the following work:


- Chapter 5 is about Constraint Answer Set Programming (CASP). We formally define the formalism, and also describe an implementation of CASP inside a lazy clause generation solver. To the best of our knowledge, this is unique from the perspective that all previous implementations of CASP are either translation based or
extensions of ASP systems. Parts of this chapter have appeared in the following paper:


- Chapters 6, 7 and 8 provide a formal description of BFASP, algorithms for solving BFASP, and the language of BFASP respectively. Parts of the first two chapters have appeared in:


Parts of the language chapter have been presented in:


- Chapter 9 provides an efficient algorithm for solving #ASP problem and applying it to probabilistic logic programming. We first discuss the obstacles in building a stable model counter when extending existing solvers or model counters. We then lay down some definitions, prove some useful properties of these newly defined entities, and show how they can help overcome those obstacles. Next, we describe how our stable model counter fits into a probabilistic logic programming solving framework to improve solving inference tasks on logic programs that involve dense positive recursion. This chapter has already been presented in the following paper:

• Chapter 10 is about \#∃ASP. We look at existing algorithms for \#∃SAT and propose two novel ideas. We also show how these algorithms can be extended to solve \#∃ASP. This work without the extension to \#∃ASP also appears in the paper:


• We conclude the thesis in Chapter 11.
Chapter 2
Formal Background

2.1 Decision Variables And Domains

For all the declarative formalisms introduced in Chapter 1, a decision variable, or simply a variable, is an entity that can take up a value from a possible domain of values. In this thesis, we denote the set of variables with $\mathcal{V}$. $\mathcal{V}$ can be decomposed in the following disjoint sets of variables, each representing a different type of domain for a variable: Boolean ($\mathcal{V}_B$) variables with domain \{true, false\}, integer variables ($\mathcal{V}_Z$) with domain $\mathbb{Z}$ and real ($\mathcal{V}_R$) variables with domain $\mathbb{R}$. For this thesis, we assume that the infinities are included in the sets of integers and real numbers. Where the distinction between integer and real variables is irrelevant in the context, we collectively represent them as the set of numeric variables $\mathcal{V}_{\text{Num}}$. The integer and real domains are total-ordered. For Booleans, we define the order as: true > false. We use the range notation \{l . . . u\} to represent the set of all values between and including $l$ and $u$. We also use the interval notation $[l, u]$ at times to represent \{l . . . u\}.

The set of variables can also be divided on the basis of foundedness type into two sets: founded variables $\mathcal{V}_F$ and non-founded, or simply, standard variables $\mathcal{V}_S$.

We often represent a group of variables $x_1, \ldots, x_n$ as $\bar{x}$.

For each formalism that we discuss in this chapter, we describe its type restrictions. Furthermore, to keep the notation simple and consistent, we remind the reader of type restrictions on $\mathcal{V}$ at the beginning of chapter where necessary (instead of using a subset of $\mathcal{V}$, e.g. $\mathcal{V}_B$ throughout the chapter).

Another important point about the notation is that mostly, we only talk about a single
program of a specific formalism, and a single global set of variables $\mathcal{V}$. But often, especially when discussing formal properties, we need to discuss more than one program and set of variables. In all such cases, we explicitly bring up this distinction, e.g. $P_1$, $P_2$ are programs with variables $\mathcal{V}_1$, $\mathcal{V}_2$ respectively etc.

### 2.2 Constraint Programming

A Constraint Satisfaction Problem (CSP) is a triple $(\mathcal{V}, D, C)$, where $\mathcal{V}$ is the set of variables, and all variables are standard, i.e. $\mathcal{V} \subseteq \mathcal{V}_S$, $D$ represents the domain of the variables, and $C$ is a set of constraints.

For each variable $v \in \mathcal{V}$, its domain $D(v)$ maps the variable to a set of values that it can take. If a variable $v$ is integer, Boolean or a real number, then $D(v) \subseteq \mathbb{Z}$ or $D(v) \subseteq \{true, false\}$ or $D(v) \subseteq \mathbb{R}$ respectively. A domain $D$ is stronger than another domain $D'$, written $D \sqsubseteq D'$, iff for all $v \in \mathcal{V}$, $D(v) \subseteq D'(v)$. A domain $D$ is strictly stronger than another domain $D'$, written $D \sqsubset D'$, iff $D \sqsubseteq D'$, but it is not true that $D' \subseteq D$. We also say that $D$ is a contraction of $D'$, or that $D$ is implied by $D'$. Two domains $D_1$ and $D_2$ are equivalent modulo a set of variables $W \subseteq \mathcal{V}$, written $D_1 =_W D_2$ iff for all $v \in W$, $D_1(w) = D_2(w)$. A domain $D$ is a false domain if $D(v) = \emptyset$ for at least one variable $v \in \mathcal{V}$. For a variable $v$ and a domain $D$, at times when doing an equality and inequality comparison of $D(v)$ with a singleton, we abuse the notation and drop the curly braces for the singleton. For example, $D(v) = false$ and $D(v) \neq 3$ mean $D(v) = \{false\}$ and $D(v) \neq \{3\}$ respectively.

The boundaries of the domains are frequently used in the solving algorithms, therefore, let us give them names. Let $lb(x)$ and $ub(x)$ give the current lower bound and upper bound respectively of $x$ in $D$. Let $val(x)$ be the unique element in $D(x)$ if $D(x)$ is a singleton and undefined otherwise. For a sequence of variables $\vec{x} = (x_1, \ldots, x_n)$, we write $D(\vec{x})$ as a shorthand for $(D(x_1), \ldots, D(x_n))$.

An assignment $\theta$ is a domain which assigns a unique value to a variable from its initial domain; i.e., $|\theta(v)| = 1$ for all $v \in \mathcal{V}$. Technically $\theta(v)$ represents a set, but as described above, we abuse it to refer to the single element inside the set, instead of the set itself.
We also call a domain a partial assignment (indicating that it is an assignment for the variables that are fixed in them).

In a CSP, we also have a set of constraints $C$. A constraint $c$ restricts the values that a set of variables, $\text{vars}(c) \subseteq V$ can be simultaneously assigned. A constraint could be specified intensionally, e.g. $\text{vars}(c) = \{x, y\}$ and $c(x, y) = x + y \leq 2$, or it could be specified extensionally, e.g. let $D(x) = \{1, 2, 3\}$, $D(y) = \{-1, 0, 1, 2\}$, $\text{vars}(c) = \{x, y\}$ and $c(x, y) = \{(1, -1), (1, 0), (1, 1), (2, -1), (2, 0), (3, -1)\}$. When written intensionally, we mostly drop the argument list of $c$ and infer $\text{vars}(c)$ by simply parsing the expression.

An assignment $\theta$ satisfies a constraint, written $\theta \models c(x)$, iff $\theta(x) \in c$. An assignment satisfies a set of constraints iff it satisfies all the constraints in the set. We call a satisfying assignment a solution.

**Definition 2.1 (Trivial constraints).** A constraint is c trivial iff it is satisfied by all possible combinations of values in the initial domains of its variables. It is non-trivial, written $\text{ntrivial}(c)$ iff it is not trivial.

Constraint Programming (CP), broadly refers to a collection of formal concepts, solving algorithms and languages developed for efficiently modelling and solving CSPs. Very often, in the problem description, there is also a cost function, $O : \theta \mapsto \mathbb{R}$, which maps an assignment to a cost, and the goal is to find a solution $\theta$ that minimizes or maximizes this cost function. Although the optimization version seems like a different problem from satisfaction, in practice, it is often solved by repeatedly solving a CSP, where after every solution, we add a constraint saying that value of the objective should be better than what we have already seen.

### 2.3 Boolean Satisfiability Problem (SAT)

In a Boolean Satisfiability Problem, or Propositional Satisfiability (SAT) problem, all variables are standard Booleans, i.e., $\mathcal{V} = \mathcal{V}_B$ and $\mathcal{V} = \mathcal{V}_S$. SAT is essentially a special case of CSPs where all variables are Booleans. A set of SAT constraints is often given in Conjunctive Normal Form (CNF) (which we explain below). All propositional formulas can be efficiently translated to CNF while preserving satisfiability, but by adding more variables.
Let us introduce some further notation. Each $v \in V$ is (also) a positive literal, and $\neg v, v \in V$ is a negative literal. Negation of a literal, $\neg l$ if $l = v$, and $v$ if $l = \neg v$. Let $\text{var}(l)$ represent the variable of the literal, i.e., $\text{var}(v) = \text{var}(\neg v) = v$. In SAT, an alternative way to represent an assignment $\theta$ is as a set of literals which represents the literals which are true in the assignment, where $\forall v \in V, \{v, \neg v\} \not\subseteq \theta$. We use this representation of assignment in the context where we have only Boolean variables. If $o$ is a formula or assignment, let $\text{vars}(o)$ be the subset of $V$ appearing in $o$. Given an assignment $\theta$, let $\theta^+ = \{v \in \theta \mid v \in V\}$ and $\theta^- = \{\neg v \in \theta \mid v \in V\}$. Given an assignment $\theta$, we use $\neg \theta$ to represent the clause $\bigvee_{l \in \theta} \neg l$.

A clause is a set of literals that represents their disjunction, we shall write as $(l_1 \lor \ldots \lor l_n)$ (and sometimes without the parenthesis where they are not necessary). A formula $F$ in conjunctive normal form (CNF) is a conjunction of clauses.

2.4 Answer Set Programming (ASP)

In ASP, $V = V_B$ and the variables are all founded (shortly we include standard variables). An ASP program $P$ is a tuple $(V, R)$ where $R$ is a set of rules $r$ of form: $a \leftarrow b_1 \land \ldots \land b_n \land \neg c_1 \land \ldots \land \neg c_m$. Let the head $a$ of the rule $r$ be denoted as $\text{head}(r)$ and the conjunctive body of $r$ as $\text{body}(r)$. Let the set of positive and negative literals in the body be given by $r^+$ and $r^-$ respectively. Let the rules of the program $R$ be also represented as $\text{rules}(P)$.

For a given variable $a \in V_F$, let $\text{rules}(a)$ be the set of rules $r$ where $a$ is the head, i.e. $\text{rules}(a) = \{r : r \in \text{rules}(P), a = \text{head}(r)\}$.

A rule is positive if its body only contains positive founded literals. The least assignment of a set of positive rules $R$, written $\text{Least}(R)$ is one that that satisfies all the rules and contains the smallest set of positive literals.

Example 2.1. Consider the ASP program $P$ with variables $a, b, c, d, e, f, g$ and a set of rules $R$ containing the following rules:
2.4 Answer Set Programming (ASP)

\[
\begin{align*}
& a \quad b \leftarrow a \quad c \leftarrow a \\
& d \leftarrow c \land a \quad e \leftarrow f \land g \quad g \leftarrow d \land f
\end{align*}
\]

All rules are positive, and \( \text{Least}(R) = \{a, b, c, d, \neg e, \neg f, \neg g\} \).

Given an assignment \( \theta \) and a program \( P \), the Gelfond-Lifschitz reduct (Gelfond and Lifschitz, 1988b) of \( \theta \) w.r.t. \( P \), written, \( P^\theta \) is a set of positive rules that is obtained as follows: for every rule \( r \), if any \( c_i \in \theta \), then \( r \) is discarded, otherwise, all negative literals are removed from \( r \) and it is included in the reduct. An assignment \( \theta \) is a stable assignment of a program \( P \) iff \( \theta = \text{Least}(P^\theta) \).

**Example 2.2.** Consider the ASP program \( P \) with variables \( a, b, c \) and a set of rules \( R \) containing the following rules:

\[
\begin{align*}
& a \leftarrow b \land \neg c \quad b \leftarrow a \land \neg c \quad c \leftarrow \neg a \land \neg b
\end{align*}
\]

Consider an assignment \( \theta = \{\neg a, \neg b, c\} \). \( P^\theta \) only contains the positive rule \( c \leftarrow \) or simply \( c \). Since \( \theta = \text{Least}(P^\theta) \), it is a stable assignment of \( P \). Now consider the exact opposite assignment \( \theta_1 = \{a, b, \neg c\} \), the reduct \( P^{\theta_1} \) contains two positive rules: \( a \leftarrow b \) and \( b \leftarrow a \). We can see that \( \text{Least}(P^{\theta_1}) = \{\neg a, \neg b, \neg c\} \), which is not equal to \( \theta_1 \), and hence \( \theta_1 \) is not a stable assignment of the program.

However, if we use propositional semantics, then \( \theta_1 \) is a satisfying assignment since it satisfies the following equivalent clauses that we get by translating each rule into its clausal equivalent:

\[
\begin{align*}
& (\neg b \lor c \lor a) \quad (\neg a \lor c \lor b) \quad (a \lor b \lor c)
\end{align*}
\]
2.4.1 Constraints And Standard Variables

In the ASP literature, a clause $b_1 \lor \ldots \lor b_j \lor \neg c_1 \lor \ldots \lor \neg c_k$ is represented as an integrity constraint, which is equal to the rule: $false \leftarrow \neg b_1 \land \ldots \land \neg b_j \land c_1 \land \ldots \land c_k$ (mostly $false$ is dropped and is implicit). The intuition is that since $false$ can never be a part of a model, therefore, at least one of the literals in the body must be false. Another way is to introduce a variable $f$, and transform all integrity constraints as the one above to: $f \leftarrow \neg b_1 \land \ldots \land c_k \land \neg f$. The idea is that $f$ can never be true in any stable model because $f$ derives $\neg f$ (contradiction): if it is true, then all its rules are eliminated in the reduct due to $\neg f$ in their bodies, and the absence of any rule means $f$ should be false. Therefore $f$ must be false, which means that all constraints must be enforced, since if any one is violated, then it derives $f$, leading to a contradiction. We use the clausal form in this thesis, and use a separate set of constraints represented by $C$. At times, when referring to some ASP program from some other source, we also use the integrity constraint form.

In classical stable model semantics (Gelfond and Lifschitz, 1988b), there is no distinction between standard (propositional) and founded variables, all variables, or atoms, are founded. Our standard variables are essentially choice variables in ASP, which have well-defined semantics (Simons et al., 2002, Simons, 2000). For a rule $r$, we define $r^+$ and $r^-$ to be the variables of literals in sets of positive and negative founded literals in $\text{body}(r)$ respectively.

With the above modifications, we can represent ASP program $P$ as a triple $(\mathcal{V}, R, C)$ where $C$ is a set of constraints (let it be also represented as $\text{cons}(P)$), and the variables could be standard or founded. This representation is not merely another point of view, but relates to SAT-based implementations (that we describe in Chapter 3) and some formalisms such as CASP much more naturally, which is why we adopt it in this thesis.

The semantics is now as follows. First, the heads of rules must be strictly founded variables. Secondly, given an assignment $\theta$ and a program $P$, $P^\theta$ is a set of positive rules that is obtained as follows: for every rule $r$, if any $c_i \in \theta$, or $\neg b_j \in \theta$ for any standard positive literal, then $r$ is discarded, otherwise, all negative literals and standard variables are removed from $r$ and it is included in the reduct. Let us write the reduced rule as $r^\theta$. An assignment $\theta$ is a stable assignment of a program $P$ iff it satisfies all its constraints
and $\theta = \nu_f \text{Least}(P^\theta)$.

The above semantics adds the treatment of standard variables and constraints as follows:

1. Handling standard variables in a similar way as negative literals in reduced rules; i.e., not including the rule in the reduct if one of them is violated in the assignment, and getting rid of them in the reduced rule if it is included.

2. Including satisfaction of all constraints by the assignment as a condition for being a stable assignment.

3. Restricting the least model equivalence to founded variables only.

**Example 2.3.** Consider the ASP program $P$ with founded variables $a, b, c$, standard variable $s, t$ the following set of rules and constraints:

$$
a \leftarrow b \land \neg c \quad b \leftarrow a \land \neg c \quad c \leftarrow \neg a \land \neg b \quad b \leftarrow s
$$

$$
a \leftarrow \neg s \land t \quad (s \lor t \lor b)
$$

Now consider the following four assignments:

$$
\theta_1 = \{ a, b, \neg c, \neg s, t \}
$$

$$
\theta_2 = \{ a, b, \neg c, s, \neg t \}
$$

$$
\theta_3 = \{ a, b, \neg c, s, t \}
$$

$$
\theta_4 = \{ \neg a, \neg b, c, \neg s, \neg t \}
$$

The reader can verify that the program has only three stable assignments: $\theta_1, \theta_2$ and $\theta_3$. However, if we exclude the last constraint from the program, then $\theta_4$ is also a stable assignment.
2.4.2 Dependency Graph

The dependency graph of a program is defined over its founded variables. For every rule and for every positive (negative) founded literal in its body, there is an edge marked positive (negative) from the head to the variable of that literal. A program is stratified if all the edges inside any given strongly connected component (SCC) in its dependency graph are positive.

The positive dependency graph of a program is the subgraph obtained by removing all negative edges from the graph above.

Example 2.4. Consider the ASP program from Example 2.3. The dependency graph has nodes $a, b, c$ and the following edges: $E = \{(a, b), (b, a), (c, a), (c, b), (a, c), (b, c)\}$. The first two edges are positive while the last four are negative. There is only one SCC and the program is not stratified due to presence of negative edges inside an SCC. But consider that we change the first two rules to: $a \leftarrow b$ and $b \leftarrow a$, then the last two edges in $E$ are eliminated, we get two SCCs: $\{a, b\}$ and $\{c\}$ and the program becomes stratified since there does not exist a negative inside any SCC.

The positive dependency graph includes these edges: $\{(a, b), (b, a)\}$.

2.4.3 Clark Completion

The Clark completion (Clark) of a logic program is a propositional formula that we get as follows. First for every variable $v \in \mathcal{V}_F$, we create a single rule which is a disjunction of all its rule bodies, and then, we turn the if condition of that rule ($\leftarrow$) to an if and only if condition ($\leftrightarrow$). It specifies that a founded variable is equal to the disjunction of all its rule bodies. For a certain class of logic programs, known as tight programs that we introduce in the next subsection, the stable assignments are the same as the propositional models of their completion plus their original constraints.

Let us formalize completion. Given a variable $a \in \mathcal{V}_F$, the completion of its rules, $\text{Comp}(a)$, is the propositional formula $a \leftrightarrow \bigvee_{r \in \text{rules}(a)} \text{body}(r)$. The completion of the entire ASP program $P$ is the propositional formula:

$$\text{Comp}(P) \equiv \bigwedge_{a \in \mathcal{V}_F} \text{Comp}(a)$$
Example 2.5. For the ASP program in Example 2.3, here are the completion formulas for all founded variables:

\[
\begin{align*}
\text{Comp}(a) &\equiv a \leftrightarrow (b \land \neg c) \lor (\neg s \land t) \\
\text{Comp}(b) &\equiv b \leftrightarrow (a \land \neg c) \lor (s) \\
\text{Comp}(c) &\equiv c \leftrightarrow (\neg a \land \neg b)
\end{align*}
\]

The completion of the entire program is:

\[
\text{Comp}(P) \equiv \text{Comp}(a) \land \text{Comp}(b) \land \text{Comp}(c)
\]

The propositional models for completion plus the original clauses, i.e., for \(\text{Comp}(P) \land (s \lor t \lor b)\), are:

\[
\begin{align*}
\theta_1 &= \{ a, b, \neg c, \neg s, t \} \\
\theta_2 &= \{ a, b, \neg c, s, \neg t \} \\
\theta_3 &= \{ a, b, \neg c, s, t \} \\
\theta_5 &= \{ a, b, \neg c, \neg s, \neg t \}
\end{align*}
\]

The first three assignments, \(\theta_1\), \(\theta_2\), and \(\theta_3\) are also the stable assignments as already described in Example 2.3. \(\theta_5\), however, is a propositional model, but not a stable assignment. \(P^{\theta_5}\) has two rules: \(a \leftarrow b\) and \(b \leftarrow a\) for which the least assignment sets every founded variable to false, which means it is not equal to \(\theta_5\) modulo the set of founded variables, since it includes \(a, b\).

It is easy to see that expanding the disjunction in the completion of a variable can give an exponentially larger CNF compared to the following alternative that introduces intermediate standard variables to represent the bodies of the rules. For a rule \(r\), introduce a variable \(b\) and post the constraint \(b \leftrightarrow \text{body}(r)\) (as an optimization, only introduce an auxiliary variable for a body if it has more than one literal; and another optimization is to not introduce a variable for a body if its head only has a single rule in the program).
Then use \( b \) instead of \( \text{body}(r) \) in the completion of \( \text{head}(r) \). Let the set of these introduced variables be equal to \( \text{bodies}(P) \).

**Example 2.6.** Let us introduce auxiliary body variables for the completion in Example 2.5. Let \( b_1 \leftrightarrow (b \land \neg c), b_2 \leftrightarrow (a \land \neg c), \) and \( b_3 \leftrightarrow (\neg s \land t) \). There is no need to introduce an auxiliary variable for the rule \( b \leftarrow s \), since it only has a single literal in the body. There is also no need to introduce a variable for the rule \( c \leftarrow (\neg a \land \neg b) \) since \( c \) has a single rule. We get the following CNF by combining the completion clauses with \( \text{cons}(P) \):

\[
\begin{align*}
\neg b \lor c \lor b_1 & \land \neg b_1 \lor \neg c & \text{ % expanding } b_1 \leftrightarrow (b \land \neg c) \\
\neg a \lor c \lor b_2 & \land \neg b_2 \lor \neg c & \text{ % expanding } b_2 \leftrightarrow (a \land \neg c) \\
s \lor \neg t \lor b_3 & \land \neg b_3 \lor \neg s & \text{ % expanding } b_3 \leftrightarrow (\neg s \land t) \\
\neg a \lor b_1 \lor b_3 & \land \neg b_1 \lor a & \text{ % expanding } a \leftrightarrow (b_1 \lor b_3) \\
\neg b \lor b_2 \lor s & \land \neg b_2 \lor b & \text{ % expanding } b \leftrightarrow (b_2 \lor s) \\
c \lor a \lor b & \land \neg c \lor \neg a & \text{ % expanding } c \leftrightarrow (\neg a \land \neg b) \\
s \lor t \lor b & \text{ % original clause}
\end{align*}
\]

\[\square\]

### 2.4.4 Tight Logic Programs

We say that a logic program is **tight** if it does not have any positive recursion. More formally, all strongly connected components of the positive dependency graph include exactly one node.

As mentioned previously, a useful property of tight programs is that their stable models coincide with the propositional models of their completion plus the constraints. Thus, we could use a propositional satisfiability solver in case of tight logic programs to find their stable assignments.
2.4.5 Unfounded Sets

A crucial concept used in ASP solving is that of unfounded sets (Van Gelder et al., 1988).

Given a partial assignment \( \theta \), a set \( U \subseteq \mathcal{V}_F \) is **unfounded** with respect to \( \theta \) iff for every rule \( r \in \text{rules}(P) \):

1. \( \text{head}(r) \notin U \), or

2. \( \theta(l) = \text{false} \) for some literal \( l \) in \( \text{body}(r) \) or

3. \( r^+ \cap U \neq \emptyset \).

Basically, a set is unfounded if every variable in it depends on some other variable in it being true, but none of them have external support, i.e. none of them can be proven true without depending on other default variables in the set. This is expressed most directly by the third alternative. The previous alternatives cover two different uninteresting cases: rules whose heads are not in the potentially unfounded set we are testing, and rules whose bodies are known to be false.

**Example 2.7.** For the program in Example 2.3, consider removing the last constraint and the partial assignment \( \theta = \{ \neg s, \neg t \} \). For this modified program, let us see if \( U = \{ a, b \} \) is an unfounded set w.r.t. \( \theta \). For the rules \( a \leftarrow b \land \neg c \) and \( b \leftarrow a \land \neg c \), the third condition applies, i.e., \( U \) intersects with the positive literals in the body. For the rules \( a \leftarrow \neg s \land t \) and \( b \leftarrow s \), the second condition applies, while for the rest of the rules, the first condition applies. Therefore, \( U \) is an unfounded set w.r.t. \( \theta \).

The greatest unfounded set w.r.t. an assignment \( \theta \) is the union of all unfounded sets w.r.t. \( \theta \). Looking at the conditions in the definition of unfounded sets above, it can be argued that the greatest unfounded set is also an unfounded set w.r.t. \( \theta \). Say the greatest unfounded set is \( G \), then for all rules \( r \in \text{rules}(P) \) where \( \text{head}(r) \in G \), one of the last two conditions must be true since we are given that there exists unfounded set \( U \subseteq G \) w.r.t. \( \theta \), which means that either some literal in the body is false in \( \theta \), or \( r^+ \cap U \neq \emptyset \) which implies that \( r^+ \cap G \neq \emptyset \). Therefore, \( G \) is also an unfounded set.
Loop Formulas

Unfounded sets of variables can be set to false, since they can never be true in an assignment that extends the assignment w.r.t. which they are unfounded, without violating the stable model semantics. Furthermore, we can encode the reason behind the unfounded set of variables being false as a propositional set of clauses. These clauses are called loop formulas (Lin and Zhao, 2004), and the intuition is that given a set of variables $U$, if all external rules are false, then the variables in $U$ must be false. Let us make this more concrete.

First, given a set of variables $U$, define the external rules of $U$ as $\text{ext}(U) = \{r : r \in \text{rules}(P), r^+ \cap U = \emptyset\}$. In other words, external rules of a set of founded variables $U$ are rules whose positive literals do not intersect with $U$. Then the loop formula says that if all external rule bodies are false, then all variables in $U$ must also be false, or equivalently, if one of the variables in $U$ is true, then at least one of the external rules must be true. This is given by:

$$\bigvee_{u \in U} u \rightarrow \bigvee_{r \in \text{ext}(U)} \text{body}(r)$$

We can break the left hand side disjunction and post a separate loop formula for each atom in a set $U$. Let $\text{loop nogood}(U, u)$ (only defined for $u \in U$) be the formula: $u \rightarrow \bigvee_{r \in \text{ext}(U)} \text{body}(r)$.

A crucial result related to loop formulas is given by Theorem 1 in Lin and Zhao (2004). The theorem states that if we take the union of program’s completion, and loop formulas for all sets of variables that form a cycle in the positive dependency subgraph, then the classical model of this propositional theory is the same as set of stable assignments of the logic program. Unfortunately, the number of such non-trivial loops in the positive dependency subgraph can be exponential in the worst case, as explained by Lifschitz and Razborov (2006).

**Example 2.8.** The spurious model $\theta_5$ in Example 2.5 is eliminated as soon as we add the loop formulas for the only cycle in the positive dependency graph ($\{a, b\}$). We reuse the auxiliary body variables introduced in Example 2.6. The set of external bodies for the set $U = \{a, b\}$ is $\{b_3, s\}$. 
The loop nogoods are: \((-a \lor b_3 \lor s)\) and \((-b \lor b_3 \lor s)\). Assuming we add \(-b_3\) to \(\theta_5\) (through unit propagation), it violates both nogoods and hence is eliminated as a propositional model of the completion plus loop formulas.

2.5 Aggregates In ASP

In this section, we review some of the aggregates commonly supported in ASP languages that are relevant to the thesis (Simons, 1999). In the first subsection, we look at cardinality and weight constraint rules. Cardinality constraint rules are a special case of weight constraint rules, where the weights or co-efficient of all variables are equal to 1. In the next subsection, we review two more aggregates: \(\min\) and \(\max\), we show that these can efficiently (linearly) be translated to simple cardinality constraints, and therefore, do not require any special attention during solving.

2.5.1 Cardinality And Weight Constraint Rules

Weight constraint rules have the following form:

\[
a \leftarrow w[b_1 = w_1, \ldots, b_j = w_j, \neg c_1 = w_{j+1}, \ldots, \neg c_k = w_{j+k}]
\]

where \(w\) is a non-negative integer, all \(w_i\)'s are positive integers and \(a \in \mathcal{V}_F\). We refer to \(w\) as the lower bound of the weight constraint rule. The cardinality constraint rule is a special case where \(w_i = 1\) for all \(i \in 1 \ldots j + k\). A cardinality constraint rule is also written as:

\[
a \leftarrow w\{b_1, \ldots, b_j, \neg c_1, \ldots, \neg c_k\}
\]

An intuitive way to write the weight constraint rule is as follows:

---

2 For the aggregates, we use the semantics described by (Simons, 1999). The discussion on alternative semantics, such as the ones by Pelov et al. (2007), Faber et al. (2011) and Gelfond and Zhang (2014), is not the focus of this thesis.

3 The ASP grounder GRINGO also translates them away, tested at least for GRINGO 3.0.4.
The $\cdot$ represents multiplication of a literal with an integer coefficient, and is equal to the coefficient when the literal is true, and 0 when the literal is false. The $l_i = w_i$ (or $l_i.w_i$ in the alternative representation above) is also called a weight literal.

The semantics is defined as follows. First, a positive program is one where there are no negative literals in weight constraint rules. Secondly, for a positive set of weight constraint rules $P$, the least assignment $\text{Least}(P)$ is one that satisfies $P$ and assigns true to the least number of variables. Thirdly, the GL-reduct is generalized for a given assignment $\theta$ as follows: for all rules $r$ with lower bound $w$, and for all weight literals $l_i = w_i$ in them, if $l_i$ is a negative literal or the variable of $l_i$ is a standard variable, then if $l_i \in \theta$, then we subtract $w_i$ from the lower bound of the rule (otherwise, we subtract 0). In other words, we subtract the weights of all negative and standard literals, that are satisfied by $\theta$, from $w$ (if the revised lower bound after subtraction is negative, we set it to 0). Finally, $\theta$ is a stable assignment $\theta =_{\forall\theta} \text{Least}(P^{\theta})$.

**Example 2.9.** Consider the following ASP program $P$ with founded variables $a, b, c$, standard variable $s$ and the following rules:

$$
a \leftarrow (b_1.w_1 + \ldots + b_j.w_j + (\neg c_1).w_{j+1} + \ldots + (\neg c_k).w_{j+k}) \geq w
$$

Consider the assignment $\theta_1 = \{a, b, \neg c, s\}$. The reduct $P^{\theta}$ is the following program with positive weight constraint rules:

$$
a \leftarrow 1[b = 2] \quad b \leftarrow 0[a = 2] \quad c \leftarrow 7[
$$

Note that the revised lower bound for the second rule is $-3$, therefore, we set it to 0. The empty weight literal body for the third rule $[\]$ represents 0. Using the intuitive representation, we can rewrite the reduct as:
\[ a \leftarrow 2. b \geq 1 \quad b \leftarrow 2. a \geq -3 \quad c \leftarrow 0 \geq 7 \]

The least assignment that satisfies the reduct is: \( \text{Least}(P^{\theta_1}) = \{a, b, \neg c\} \) which is equal to \( \theta_1 \) modulo founded variables, and hence \( \theta_1 \) is a stable assignment of \( P \).

Consider \( \theta_2 = \{a, b, \neg c, \neg s\} \). The reduct is:

\[ a \leftarrow 1[b = 2] \quad b \leftarrow 2[a = 2] \quad c \leftarrow 7[\]

The least assignment for the reduct is: \( \text{Least}(P^{\theta_2}) = \{\neg a, \neg b, \neg c\} \) which is not equal to \( \theta_2 \) modulo founded variables, and hence \( \theta_2 \) is not a stable assignment of \( P \).

\[
2.5.2 \quad \text{Minimum And Maximum Aggregates}
\]

ASP languages also support two other aggregates, minimum and maximum, also defined with weight literals. We review how they can be translated in terms of previously defined rule forms. Let us look at minimum aggregate first. It has the form:

\[
a \leftarrow \min \ w[b_1 = w_1, \ldots, b_j = w_j, \neg c_1 = w_{j+1}, \ldots, \neg c_k = w_{j+k}]
\]

Let us define \( l_i = b_i \) if \( i \in 1 \ldots j \) and \( l_i = \neg c_{i-j} \) if \( i \in j + 1 \ldots j + k \). Then the above rule can be written as:

\[
a \leftarrow \min \ w[l_1 = w_1, \ldots, l_{j+k} = w_{j+k}]
\]

The intuition for a rule with minimum aggregate is that given an assignment, among all weight literals whose Boolean literal is in the assignment, pick the one with minimum weight \( w_i \); if \( w_i \geq w_j \), then the body evaluates to true. If no literal is true in the assignment, the minimum value is \( \infty \), and the body evaluates to true. The above rule can be replaced with the rule \( a \leftarrow \neg x \) and the following cardinality constraint rule that defines when \( x \) becomes true.
\[ x \leftarrow 1\{l_i : i \in 1 \ldots j + k, w_i < w\} \]

The cardinality constraint rule models that the only way that the aggregate body is false (\(x\) is true) is if one of the weight literals with weight strictly less than \(w\) is true.

Rules with maximum aggregates have the form:

\[ a \leftarrow \max w[b_1 = w_1, \ldots, b_j = w_j, \neg c_1 = w_{j+1}, \ldots, \neg c_k = w_{j+k}] \]

The intuition is that given an assignment, among all weight literals whose Boolean literal is in the assignment, pick the one with maximum weight \(w_i\); if \(w_i \geq w\), then the body should evaluate to true. If no literal is true in the assignment, the maximum value is \(-\infty\), and the body evaluates to false. We can replace it with the rule \(a \leftarrow x\) and the cardinality constraint:

\[ x \leftarrow 1\{l_i : i \in 1 \ldots j + k, w_i \geq w\} \]

### 2.6 Fuzzy Answer Set Programming

Fuzzy Answer Set Programming (FASP) is a generalization of ASP. In FASP, in addition to Boolean variables, some variables can take on fuzzy or partially true values. Essentially, this means that these are variables with a continuous domain \([0, 1]\). The negation of a fuzzy value \(\neg v\) of a variable \(v\) is equal to \((1 - v)\). The rules in FASP are of the form:

\[ a \leftarrow b_1 \otimes \ldots \otimes b_n \otimes \neg c_1 \otimes \ldots \otimes \neg c_m \]

where \(\otimes\) is one of Gödel, Product, or Lukasiewicz t-norm, defined for \(x \otimes y\) as follows:

- **Gödel:** \(\min(x, y)\)
- **Product:** \(x.y\)
- **Lukasiewicz:** \(\max(0, x + y - 1)\)
It can be seen that all the above norms are commutative \((x \otimes y = y \otimes x)\), monotonic \((x \otimes y \leq x' \otimes y' \text{ if } x \leq x' \text{ and } y \leq y')\), associative \((x \otimes (y \otimes z) = (x \otimes y) \otimes z)\), and have 1 as the identity element.

The semantics is defined as follows. First, for a program with only positive rules (rules that do not have any negative literal), an assignment \(\theta\) is minimum iff \(\theta\) satisfies the program and there is no other assignment \(\theta'\) such that \(\theta'\) also satisfies the program and \(\theta'(v) < \theta(v)\) for at least one variable \(v\).

Next, given an assignment \(\theta\), the reduct of the program w.r.t. \(\theta\) is defined as replacing all negative literals with their values in \(\theta\). The assignment is a stable assignment iff it is equal to the minimum assignment of the reduct.

**Example 2.10.** Consider the following FASP P:

\[
a \leftarrow b \otimes \neg c \quad b \leftarrow a \otimes \neg c \quad c \leftarrow \neg a \otimes \neg b
\]

Now say we have two assignments \(\theta_1 = \{a \mapsto 0.5, b \mapsto 0.4, c \mapsto 0.8\}\) and \(\theta_2 = \{a \mapsto 0, b \mapsto 0, c \mapsto 1\}\). Their respective reducts are:

\[
P^{\theta_1}: a \leftarrow b \otimes 0.2 \quad b \leftarrow a \otimes 0.2 \quad c \leftarrow 0.5 \otimes 0.6
\]

\[
P^{\theta_2}: a \leftarrow b \otimes 0 \quad b \leftarrow a \otimes 0 \quad c \leftarrow 1 \otimes 1
\]

Let us check whether \(\theta_1\) and \(\theta_2\) are fuzzy answer sets for all three norms. For Gödel norm, the reduct is:

\[
P^{\theta_1}: a \leftarrow \min(b, 0.2) \quad b \leftarrow \min(a, 0.2) \quad c \leftarrow \min(0.5, 0.6)
\]

\[
P^{\theta_2}: a \leftarrow \min(b, 0) \quad b \leftarrow \min(a, 0) \quad c \leftarrow \min(1, 1)
\]

The minimum assignments for \(P^{\theta_1}\) and \(P^{\theta_2}\) are \(\{a \mapsto 0, b \mapsto 0, c \mapsto 0.5\}\) and \(\{a \mapsto 0, b \mapsto 0, c \mapsto 1\}\) respectively.

\[\text{An assignment is commonly known as an interpretation in the FASP literature.}\]

\[\text{The technique to plug in truth values in the reduct for multi-valued logics can be traced back at least to Przymusinski (1991), who defined 3-valued stable models in this way.}\]
For product norm, the reduct is:

\[ P_{\theta}^1 : a \leftarrow b.(0.2) \quad b \leftarrow a.(0.2) \quad c \leftarrow (0.5).(0.6) \]

\[ P_{\theta}^2 : a \leftarrow b.0 \quad b \leftarrow a.0 \quad c \leftarrow (1).(1) \]

The minimum assignments for \( P_{\theta}^1 \) and \( P_{\theta}^2 \) are \( \{a \mapsto 0, b \mapsto 0, c \mapsto 0.3\} \) and \( \{a \mapsto 0, b \mapsto 0, c \mapsto 1\} \) respectively.

For Lukasiewicz norm, the reduct is:

\[ P_{\theta}^1 : a \leftarrow \max(0, b + 0.2 - 1) \quad b \leftarrow \max(0, a + 0.2 - 1) \quad c \leftarrow \max(0, 0.5 + 0.6 - 1) \]

\[ P_{\theta}^2 : a \leftarrow \max(0, b + 0 - 1) \quad b \leftarrow \max(0, a + 0 - 1) \quad c \leftarrow \max(0, 1 + 1 - 1) \]

The minimum assignments for \( P_{\theta}^1 \) and \( P_{\theta}^2 \) are \( \{a \mapsto 0, b \mapsto 0, c \mapsto 0.1\} \) and \( \{a \mapsto 0, b \mapsto 0, c \mapsto 1\} \) respectively.

For all norms, \( \theta_1 \) is not a fuzzy answer set since it is not equal to the minimum model of \( P_{\theta}^1 \) and \( \theta_2 \) is a fuzzy answer set of \( P \) since it is equal to the minimum model of \( P_{\theta}^2 \).

2.7 Propositional Model Counting\(^6\)

Propositional model counting, or simply model counting, is the problem of counting the number of assignments to a set of variables such that a Boolean formula defined over those variables is satisfied. It is also known as \#SAT.

We need to introduce some notation in order to deal with model counting and discuss its solving algorithms. Given an assignment \( \theta \), the residual of a CNF \( F \) w.r.t. \( \theta \) is written \( F|_{\theta} \) and is obtained by removing each clause \( C \) in \( F \) such that there exists a literal \( l \in C \cap \theta \), and simplifying the remaining clauses by removing all literals from them whose negation is in \( \theta \). We say that an assignment \( \theta \) is a solution cube, or simply a cube, of \( F \) iff \( F|_{\theta} \) is

\(^6\)This and all subsequent sections in this chapter are required preliminaries for Part II of the thesis, but not for Part I.
empty. The size of a cube $\theta$, $\text{size}(\theta)$ is equal to $2^{\mid V \mid - \mid \theta \mid}$. A solution in the classical sense is a cube of size 1. The model count of $F$, written, $\text{count}(F)$ is the number of solutions of $F$.

Example 2.11. Consider the CNF $F$ with variables $a, b, c$ and two clauses: $(a \lor \neg b \lor c)$ and $(b \lor \neg c)$. Consider the partial assignment $\theta = \{b\}$, then $F|_{\theta}$ has only one clause $(a \lor c)$ that comes from the first clause by removing $\neg b$; the other clause is satisfied by the assignment. Now, say we extend $\theta$ by $a$, i.e., $\theta = \{b, a\}$, then the residual is empty, and $\text{size}(\theta) = 2^{\mid \{a, b, c\} \mid - \mid \theta \mid} = 2^{3-2} = 2$.

Following are all models of $F$: $\{\neg a, \neg b, \neg c\}$, $\{\neg a, b, c\}$, $\{a, \neg b, \neg c\}$, $\{a, b, \neg c\}$, $\{a, b, c\}$. Since there are 5 models, $\text{count}(F) = 5$.

2.8 Knowledge Compilation

The research area of propositional knowledge compilation looks at different ways of representing a Boolean formula, and studies how they relate to one another and the kind of operations and transformations that can be performed on each one of them efficiently (Darwiche and Marquis, 2002). A representation that is relevant to this thesis is called $d$-DNNF (Darwiche, 2001), and we explain it below.

A Boolean formula is in negation normal form (NNF) iff the only sub-formulas that have negation applied to them are propositional variables. An NNF formula is decomposable (DNNF) iff for all conjunctive formulae $c_1 \land \cdots \land c_n$ in the formula, the sets of variables of conjuncts are pairwise disjoint, $\text{vars}(c_i) \cap \text{var}(c_j) = \emptyset, 1 \leq i \neq j \leq n$. Finally, a DNNF is deterministic (d-DNNF) if for all disjunctive formulae $d_1 \lor \cdots \lor d_n$ in the formula, the disjuncts are pairwise logically inconsistent, i.e. $d_i \land d_j$ is unsatisfiable for all $1 \leq i \neq j \leq n$. A d-DNNF is typically represented as a tree or DAG with inner nodes and leaves being OR/AND operators and literals respectively. Model counting on d-DNNF can be performed in polynomial time (in d-DNNF size) by first computing the satisfaction probability and then multiplying the satisfaction probability with total number of assignments. Satisfaction probability can be computed by evaluating the arithmetic expression that we get by replacing each literal with 0.5, $\lor$ with $+$ and $\land$ with $\times$ in the d-DNNF.
Example 2.12. Let $F$ be a CNF with variables $a, b, c, d, e$ and clauses $(\neg b \lor \neg a \lor c)$ and $(\neg b \lor d \lor \neg e)$. The following is a $d$-DNNF for $F$:

To get the satisfaction probability, we evaluate the following expression:

The above expression is equal to $0.5 \times (0.5 \times 0.5 + 0.5)^2 + 0.5 = 0.78125$. Therefore, $\text{count}(F) = 0.78125 \times 2^{\vert\{a,b,c,d,e\}\vert} = 25$. 

\qed
Chapter 3
Solving Algorithms

We review some well known and widely used techniques for solving CP and SAT. Next, we look at lazy clause generation, which is a hybrid CP/SAT approach that has shown great success in constraint solving. For ASP solving, we review some of the algorithms that implement ASP via detecting and pruning unfounded sets. Finally, we describe the techniques that are relevant to this thesis for propositional model counting.

3.1 Finite Domain Constraint Solving

In finite domain constraint solvers, every constraint $c$ is associated with a propagator $p_c$. A propagator is a contracting function from domains to domains. A correct propagator has two properties. First, any value for any variable that it prunes from its domain, must be infeasible. More concretely, for a given domain $D$, let $D'$ be a domain such that for each variable $v \in V$, we have that $D'(v) = D(v) \setminus p_c(D)(v)$, i.e. $D'$ contains the values that are pruned by the propagator. Then, if there is any assignment $\theta$ where $\theta(v) \in D'$ for any variable $v$ and $\theta(u) \in D(u)$ for the rest of the variables $u \in V \setminus \{v\}$, then $\theta$ must not be a solution of $c$. That is, no assignment in which at least one variable takes up a value pruned by the propagator, can be a solution. The second requirement is that the propagator must correctly check for failures when all variables are fixed. Concretely, if an assignment $\theta$ does not satisfy the constraint $c$, then $p_c(\theta)$ is a false domain. Recall from Section 2.2 that a false domain is one that has an empty domain for at least one variable. The propagator, in this case, signifies that it is impossible to assign any value to one or more variables without violating the constraint.

A constraint programming solver solves the CSP with initial domain $D$ and con-
straints $C$ by interleaving propagation with choice. It applies all propagators $p_c$ for constraints $c$ to the current domain $D$, and it does so repeatedly until no propagator makes a change (i.e. until a fixpoint is reached). If the final domain $D$ represents failure ($D(x) = \emptyset$ for some $x$), then it backtracks to try another choice. If every variable has at least one element in its domain, but at least one has two or more, then the solver needs to make a choice by splitting the domain of one of these variables into two or more parts. This decision, choice, or labelling step results in two or more subproblems which the solver then solves recursively. If all variables have exactly one value in their domains, then there are no choices left to be made, and the domain is actually an assignment. If all the propagators return a non-failure, then the assignment is a solution. In practice, solvers use event-driven scheduling of propagators and priority mechanisms to try to reach fixpoints as quickly as possible (Schulte and Stuckey, 2008). More particularly, they use domain change events such as lower/upper bound change events or value fixing events ($lb\_event(x), ub\_event(x), fix\_event(x)$ resp.) to lazily wake up propagators. Each propagator subscribes to the events that may allow it to prune additional values.

**Example 3.1.** This example has been adopted from the one by Stuckey (2014). There are five variables $x_1, x_2, x_3, x_4, x_5$, all with initial domains $1 \ldots 4$. There are three constraints:

\[
\begin{align*}
    c_1 & : \text{alldifferent}([x_1, x_2, x_3, x_4]) \\
    c_2 & : x_2 \leq x_5 \\
    c_3 & : x_1 + x_2 + x_3 + x_4 \leq 9
\end{align*}
\]

The first constraint is a global constraint, stating that the values of $x_1 \ldots x_4$ should be pairwise distinct. The second and third constraints are linear constraints.

Let us say that we have bounds propagators for the linear constraints. A bounds propagator for $c_2$ implements the following two rules:

\[
\begin{align*}
    x_2 & \leq ub(x_5) \\
    x_5 & \geq lb(x_2)
\end{align*}
\]
The first rule says that the upper bound of \( x_2 \) should be less than or equal to the upper bound of \( x_5 \). For example, if \( D(x_5) = 1 \ldots 2 \), and \( D(x_2) = 1 \ldots 4 \), then this rule propagates \( D(x_2) = 1 \ldots 2 \), i.e. bringing the upper bound of \( x_2 \) from 4 to 2. The second rule is analogous for the lower bound of \( x_5 \). Note that the first rule only needs to trigger whenever \( \text{ub}(x_5) \) is modified. Similarly, the second rule can only adjust the lower bound of \( x_5 \) whenever \( \text{lb}(x_2) \) is modified.

Therefore, the bounds propagator for \( c_2 \) implementing the above two rules should subscribe to the following events: \( \text{ub\_event}(x_5) \) and \( \text{lb\_event}(x_2) \). Note that these conditions are necessary for the propagator to produce a strictly stronger domain, but they are not sufficient. For example, if \( D(x_2) \) is already equal to \( 1 \ldots 2 \), then if the upper bound of \( x_5 \) changes from 4 to 2, then the first rule does nothing to the upper bound of \( x_2 \).

The bounds propagation rules for constraint \( c_3 \) are:

\[
\begin{align*}
x_1 &\leq 9 - \text{lb}(x_2) - \text{lb}(x_3) - \text{lb}(x_4) \\
x_2 &\leq 9 - \text{lb}(x_1) - \text{lb}(x_3) - \text{lb}(x_4) \\
x_3 &\leq 9 - \text{lb}(x_1) - \text{lb}(x_2) - \text{lb}(x_4) \\
x_4 &\leq 9 - \text{lb}(x_1) - \text{lb}(x_2) - \text{lb}(x_3)
\end{align*}
\]

This propagator should subscribe to the events: \( \text{lb\_event}(x_1), \text{lb\_event}(x_2), \text{lb\_event}(x_3), \) and \( \text{lb\_event}(x_4) \).

For the alldifferent constraint, let us say that we have a propagator that implements the simple propagation rules: \( x_i \neq \text{val}(x_j) \) for all \( i, j \in \{1, 2, 3, 4\} \) and \( i \neq j \). For given values of \( i, j \), the rule eliminates the value \( \text{val}(x_j) \) from \( D(x_i) \). Recall from Section 2.2 that \( \text{val}(x) \) is only defined when the domain of \( x \) is a singleton. This propagator should only trigger when the value of any of \( x_1, x_2, x_3, x_4 \) is fixed. Therefore, this propagator should subscribe to the events: \( \text{fix\_event}(x_1), \text{fix\_event}(x_2), \text{fix\_event}(x_3), \) and \( \text{fix\_event}(x_4) \).

Now, let us go over a possible trace that we get by executing the algorithm described above, that interleaves search with propagation. The trace is given in Table 3.1.

The first column from left has the variable names. Every column after that is either a decision, or a propagation step. There are only two decisions, \( x_1 = 1 \) and \( x_5 \leq 2 \) and the two decisions
along with the propagation steps that they prompt are separated by the vertical line. The change in the domain of each variable, caused by decision or propagation, is written in bold. For example, the first decision changes the domain of $x_1$ from $1 \ldots 4$ to $1$, and therefore $1$ is written in bold.

The second decision eventually leads to a failure, caused by the alldifferent constraint, since $x_3$ and $x_4$ have the same values. The solver, therefore, backtracks, and tries the other part of bisected domain of $x_5$, i.e., it tries $x_5 > 2$, the result of which is shown in Table 3.2. Backtracking is reflected by the deletion of all columns after the vertical line from Table 3.1 and addition of the last column with the decision $x_5 > 2$. 

\[ \begin{array}{cccccc}
\mathbf{x_1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\
\mathbf{x_2} & 1 \ldots 4 & 2 \ldots 4 & 2 \ldots 4 & 2 \ldots 4 & 2 \ldots 4 \\
\mathbf{x_3} & 1 \ldots 4 & 2 \ldots 4 & 2 \ldots 4 & 2 \ldots 4 & 2 \ldots 4 \\
\mathbf{x_4} & 1 \ldots 4 & 2 \ldots 4 & 2 \ldots 4 & 2 \ldots 4 & 2 \ldots 4 \\
\mathbf{x_5} & 1 \ldots 4 & 1 \ldots 4 & 2 \ldots 4 & 2 \ldots 4 & 2 \ldots 4 \\
\end{array} \]
3.2 SAT Solving

SAT solvers have become very powerful at solving CNF problems in the recent decades and it is not unusual for them to solve problems involving millions of Boolean variables and a similar number of clauses. In this section, we look at some of the most widely used techniques in SAT solving.

3.2.1 DPLL And Unit Propagation

State-of-the-art SAT solvers are based on the Davis-Putnam-Longemann-Loveland (DPLL) algorithm (Davis et al., 1962). DPLL is similar to the search and propagate solving algorithm described in the previous section for constraint solving. The propagation, though, is a much simpler case. The original DPLL algorithm had only two kinds of propagators; unit propagation and pure literal elimination.

Given an assignment \( \theta \) of literals, if there is a clause \((l_1 \lor \ldots \lor l_i \lor \ldots \lor l_n)\), and \(\{l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n\}\) are all in \( \theta \), then unit propagation adds \( l_i \) in \( \theta \). The idea is straightforward, in order for the clause to be satisfied, the literal \( l_i \) must be true.

Pure literal elimination means that if there is no occurrence of \( \neg l \) in all clauses, then we can safely extend the assignment by \( l \). Pure literal elimination requires that we revisit all clauses to check if they contain \( \neg l \) after every decision, which can be expensive. This propagation is rarely used in modern SAT solving, especially since the idea of watched literals alleviates the need to visit each clause eagerly.

Let us look at an example of DPLL with unit propagation on a CNF.

Example 3.2. Consider the following CNF problem with variables \( a, b, c, d, e \):

\[
(\neg a \lor \neg e \lor c \lor \neg d) \quad (b \lor \neg c \lor \neg a) \quad (\neg d \lor \neg e \lor \neg c) \quad (\neg a \lor d)
\]

Figure 3.1 shows a possible trace of DPLL algorithm. Initially all variables are unfixed and unit propagation on the CNF does not prune any value. We try the decision \( a \), which unit propagates \( d \) due to the last clause. Next we try \( e \) which propagates \( \neg c \), \( b \), and \( c \). \( \neg c \) and \( c \) imply that the domain of \( c \) is empty, and this is a failure. Therefore, we backtrack, undo the decision \( e \) and try
\[ \neg e \text{ instead, which does not propagate anything. We make another choice } \neg b, \text{ which propagates } c. \] Since all variables are fixed and the assignment is consistent, \( \{a, d, \neg e, \neg b, c\} \) is a satisfying assignment.

\[ \square \]

### 3.2.2 Watched Literals

An important implementation technique for DPLL uses watched literals. Watched literals are motivated by the fact that unit propagation is triggered when all literals in a clause except for one are set to false. Therefore, as long as we have two non-false literals in a clause, unit propagation cannot yield anything. The SAT solver keeps a list of clauses against each literal, which represents the watched clauses for the literal. When a literal \( l \) becomes false, then for each clause \( c \) that is in its watched list, we check if there is an alternative literal \( l' \) in \( c \) that is non-false and is a candidate new watch. If yes, we simply update the watch of \( c \) from \( l \) to \( l' \), if no, then we unit propagate the second watched literal of \( c \). Another useful property of watched literals is that upon backtracking, we do not need to update any watch since anything that is non-false at a later stage is also non-false at an earlier decision level.
3.2.3 Conflict Analysis, Nogood Learning And Backjumping

The idea of nogood learning is to analyze a failure and derive a clause that explains the failure, and add this clause to the theory. These learned clauses are logically redundant w.r.t. the theory, but can provide additional propagation that allows the solver to avoid redundant search. A naive way to explain failure is to say that we failed because of the entire current assignment \( \theta \), and add the negation of the current assignment as a clause, i.e., \(( \lor_{l \in \theta} \neg(l))\). But, we can do much better than that.

A central concept in conflict analysis is that of implication graph. An implication graph is a DAG that records the reason for every literal in the current assignment. Decisions have no reason, and hence they have no incoming edges. The incoming edges for the literals that are unit propagated are the negative literals of the rest of the literals in the clause, that led to the propagation of that literal. Once we reach a failure, the conflict analysis is performed as follows: we keep a frontier, or cut, which is a set of literals whose conjunction provably leads to a failure. The cut is maintained as a stack where literals that are fixed latest are popped first. We start by pushing the negative of the literals of the clause that is violated. Then we resolve each literal by popping it from the frontier, and pushing its unseen ancestors in the stack. Now, there are various schemes each of which indicate when we should stop resolving. The most popular one that works perhaps best in practice is called the 1-UIP scheme and it tells us to stop as soon as there is exactly one literal from the last decision level (the decision level that led to failure) in the frontier.

Now that we have a set of literals whose conjunction leads to failure, we can infer the negation of their conjunction as a redundant formula. When we push the negation in, we get a clause that is equal to the disjunction of negative of literals in the frontier. This redundant clause is called the conflict clause.

Once we have learned the conflict clause, we could backtrack to the last decision level as usual, propagate the conflict clause and continue. But we could actually go to the earliest decision level where the conflict clause is asserting, i.e., where unit propagates on the clause extends the assignment, since there might be some intermediate decision levels.

\(^7\)The term nogood learning is commonly used in the CP community, while clause learning is used in the SAT community to refer to the same idea.
which had nothing to do with the failure. This is called backjumping. Let us demonstrate all these concepts through an example.

**Example 3.3.** Consider the following CNF with variables $a, b, c, d, e, f, g$, we also name the clauses since it vastly improves the illustration of implication graph.

$C_1: (\neg a \lor \neg b)$  
$C_2: (\neg a \lor \neg g \lor i)$  
$C_3: (b \lor d \lor c)$  
$C_4: (\neg i \lor \neg h)$  
$C_5: (b \lor \neg g \lor h)$  
$C_6: (\neg c \lor e)$  
$C_7: (f \lor g)$

Figure 3.2 shows the implication graph when we take the decisions that are shown in bold: $a, \neg d, \neg f$. Unit propagation yields more literals; the clauses that derive those literals are shown in their superscripts in parentheses. After the first decision, through unit propagation, we get $\neg b$. After the second decision, we get $c$ and $e$, and after the third decision we get $g, h, i$; the last two of which violate the clause $C_4$.

We start the conflict analysis with the cut $C = \{i, h\}$. It is intuitive to read this as: we failed because $i$ and $h$ are true. Now, if we resolve $i$, we get $\{a, g\}$, so we update $C = \{a, g, h\}$ (we fail because $a$, $g$, and $h$ are true and so on). Resolving $h$ gives $C = \{a, g, \neg b\}$ ($g$ already exists in the cut). Now we stop since there is only one literal from the last decision level, $g$. Our conflict clause is $\neg C = (\neg a \lor \neg g \lor b)$. 

![Implication Graph Figure 3.2](image-url)
3.3 Lazy Clause Generation

As clear from the graph, the second decision has nothing to do with the failure, and we can also infer this from our conflict clause. The latest decision level of any literal in \( C \), excluding \( g \) is 1. So we backjump to decision level 1 and propagate \( \neg g \).

Besides watched literals, conflict analysis, nogood learning, and backjumping, modern SAT solvers employ a number of other techniques that have proven to be very effective in practice. Most notable of these are activity-based search schemes, in which the literals that appear most in recent conflict clauses are preferred candidates for decisions; and restarts, in which the solver is restarted after a certain number of failures (that usually increases after one or more restarts meaning the restarts become less frequent). See the work by following authors for further discussion: Biere et al. (2009), Moskewicz et al. (2001), van Harmelen et al. (2007).

3.3 Lazy Clause Generation

Lazy clause generation (Feydy and Stuckey, 2009, Ohrimenko et al., 2009) is a hybrid technique that combines the power of SAT solving and CP propagation. SAT solving techniques as described in Section 3.2 have proven to be very successful. Therefore, it seems like an attractive idea to exploit the power of SAT for CP solving. However, in general, there is no efficient way of encoding CP propagation, which is at the core of efficient CP solving, as SAT clauses. The clausal representation of all possible propagation steps can be exponentially large. Let us illustrate that with the help of an example.

**Example 3.4.** Consider the following bounds propagation rule from Example 3.1 for the constraint \( c_3 \):

\[
x_1 \leq 9 - \text{lb}(x_2) - \text{lb}(x_3) - \text{lb}(x_4)
\]

One propositional representation of the above rule is to use the unary encoding to represent integer domains, and use the following propositional rules (clauses) given below. In unary encoding, to represent the domain \( l \ldots u \) for an integer variable \( x \), we have propositional variables \([x \leq l], \ldots, [x \leq u] \), and the constraints: \( \forall l \leq d \leq u - 1, \text{we have that } [x \leq d] \rightarrow [x \leq d + 1] \).
These constraints ensure a consistent domain, and avoid cases like \([x \leq 5]\) and \(\neg[x \leq 6]\). Now let us look at one way of encoding clauses for the above propagation nogoods using unary encoding.

\[
\begin{align*}
[x_1 \leq 6] & \leftarrow [x_2 \geq 1] \land [x_3 \geq 1] \land [x_4 \geq 1] \\
[x_1 \leq 5] & \leftarrow [x_2 \geq 1] \land [x_3 \geq 1] \land [x_4 \geq 2] \\
[x_1 \leq 4] & \leftarrow [x_2 \geq 1] \land [x_3 \geq 1] \land [x_4 \geq 3] \\
& \vdots \\
[x_1 \leq -3] & \leftarrow [x_2 \geq 4] \land [x_3 \geq 4] \land [x_4 \geq 4]
\end{align*}
\]

In the rules above, \([x \geq k]\) is a shorthand for \(\neg[x \leq k - 1]\). There are \(4^3\) of above such rules, showing that the number of clauses to encode (part of) a propagator can be exponential in the domain sizes of its variables.

The above example shows that \textit{a priori} translation to SAT can be very expensive. A careful reader might point out that the exponential blow up in the clausal representation is predicated on the unary encoding that we used in the example. This observation is not baseless. It might be possible to encode some propagators using a logarithmic (in the size of domain) number of clauses, but such encodings do not propagate as strongly. The discussion on propositional encodings of finite domain variables as well as propagators is beyond the scope of this thesis and we redirect the reader to Ohrimenko et al. (2009) for further discussion. We next describe the finite domain variables encoding that is used in lazy clause generation systems, and then move on to the description of how a lazy clause generation solver works.

To represent an integer \(x\) with the initial domain \([l \ldots u]\), the following \(2.(u - l + 1)\) variables are used: \([x = l], \ldots, [x = u]\) and \([x \leq l], \ldots, [x \leq u]\). Furthermore, the clausal equivalent of the following constraints are included to ensure that these variables represent a consistent domain:
∀l ≤ d ≤ u − 1 \[ x ≤ d ] \rightarrow \[ x ≤ d + 1 ]

∀l ≤ d ≤ u : \[ x = d ] \leftrightarrow ( [ x ≤ d ] \land \neg [ x ≤ d − 1 ] )

Following the original convention, we refer to the above clauses as DOM. Now, let us turn our attention to the search and propagation inside a modern lazy clause generation solver as described by Feydy and Stuckey (2009). The idea is that the finite domain propagators work like they do in a CP solver with one addition. Whenever they propagate, they post a clause explaining that propagation, using the propositional variables described above. For example, say in the instance in Example 3.1, the lower bounds of \( x_2, x_3, \) and \( x_4 \) are 2, 2, and 3 respectively and one of the lower bound events for these variables just fired. The bounds propagator now, in addition to adjusting the upper bound of \( x_1 \) to 2, posts the following clause:

\[ [ x_1 ≤ 2 ] \lor \neg [ x_2 ≤ 2 ] \lor \neg [ x_3 ≤ 2 ] \lor \neg [ x_4 ≤ 3 ] \]

In the solver, SAT unit propagation runs before all other propagators in the solver. Let us see the potential advantages of explaining the propagation as clauses. Say, later in the search, we have similar lower bounds on \( x_2, x_3, \) and \( x_4 \), then unit propagation on this clause will immediately yield \( [ x_1 ≤ 2 ] \). The solver will notice this and adjust the domain of \( x_1 \) adequately. Note that like learned nogoods, these clauses are redundant as long as we have propagators running in the solver, i.e., if we delete the above clause, the propagator would still run and adjust the upper bound of \( x_1 \) as it previously did, which means we can remove these clauses, if and when required.

Having potentially faster propagation via unit propagation is a useful by-product, but not the primary advantage of lazy clause generation. The real strength comes from the fact that the (partial) clausal representation allows us to apply all useful SAT techniques such as nogood learning, backjumping etc that we described in Section 3.2 on FD reasoning. In fact, later solvers like CHUFFED (Chu, 2011) do not store the explanation clauses for propagation but only for reasoning with implication graph and throw them away as soon as we backtrack. Let us demonstrate that by solving the CP instance in Example 3.1
Example 3.5. Figure 3.3 shows the implication graph inside a lazy clause generation solver when run with the two decisions shown in Table 3.1. The decisions are shown in boxes with thicker boundary. All propagators explain themselves. Furthermore, we have DOM clauses as described above.

The first decision is \( x_1 = 1 \). The alldifferent constraint propagates \( x_2 \neq 1 \), \( x_3 \neq 1 \), and \( x_4 \neq 1 \) and explains them as the following clauses: \( (\neg x_1 = 1 \lor \neg x_2 = 1) \), \( (\neg x_1 = 1 \lor \neg x_3 = 1) \), and \( (\neg x_1 = 1 \lor \neg x_4 = 1) \). Note that the literals \( \neg x_2 = 1 \), \( \neg x_3 = 1 \), and \( \neg x_4 = 1 \) are shown as \( x_2 \neq 1 \), \( x_3 \neq 1 \), and \( x_4 \neq 1 \) respectively for better readability. These clauses are entered into the implication graph adequately. Due to the DOM clause \( \neg x_2 = 1 \Rightarrow \neg x_2 \leq 0 \lor \neg x_2 \leq 1 \), since 0 is not in the initial domain \( x_2 \leq 0 \) is false, and therefore, through unit propagation, we get \( \neg x_2 \leq 1 \), or equivalently, \( x_2 \geq 2 \). We get similar propagations for \( x_2 \) and \( x_3 \). Due to the change in lower bound of \( x_2 \), the propagator for the constraint \( x_5 \geq x_2 \) propagates \( x_5 \geq 2 \), and explains this as \( x_2 \leq 1 \lor \neg x_5 \leq 1 \). This is recorded in the implication graph by the SAT solver.

Following similar reasoning for the second decision \( x_5 \leq 2 \), we get the implication graph as it is shown in Figure 3.3, which shows that alldifferent declares failure with the reason \( (\neg x_3 = 3 \lor \neg x_4 = 3) \). Analyzing this conflict as described in Section 3.2.3 yields the nogood: \( x_2 \leq
3.4 Unfounded Set Propagation

In ASP solving, detecting unfounded sets given a partial assignment is of critical importance. In this section, we look at some well-known unfounded set detection algorithms, and how they are used with different search algorithms for stable model computation.
3.4.1 Source Pointer Technique

The most widely used idea for unfounded set detection during search is the source pointer technique (Simons et al., 2002, Simons, 2000). It implements detection of the greatest unfounded set w.r.t. a given (partial or complete) assignment, in order to propagate all variables in the greatest unfounded set to false. To achieve that, it maintains a justification graph where there are two types of edges: one from a variable to a non-false rule, signifying that this variable has a source or a justification to be non-false; and the other from all positive variables in a rule body to the rule itself, indicating that the rule itself uses these variables as justifications to justify its head. One way to represent the justification graph, is to actively maintain all edges, but that is not necessary. Since the edges from a rule to all its positive variables do not change during the search, it is sufficient to only maintain the first set of edges, that is, edges from founded variables to rules. This is the reason why these edges are called source pointers.

The key idea is that there can never be an unfounded set as long as the graph is acyclic. Naively, we could recompute the entire justification graph from scratch each time the partial assignment is changed. Or, we could do it incrementally as follows. Given a set of variables and rule bodies that have become false since the last time that the graph was updated, the algorithm first cancels out (dejustifies) all descendants of these nodes. Then, in the second phase, it attempts to reach (rejustifies) as many nodes as possible that lost their justification during dejustification. The algorithm optimizes this by restricting the dejustification/rejustification computation to a single strongly connected component in the positive dependency graph at one time, and does this for all SCCs in the right topological order. So for example, if one variable is dejustified and rejustified as well, then the computations of descendant SCCs that would have been triggered by its dejustification do not occur.

Throughout the text, we interchangeably use justification graph and source pointers. Note that the algorithm described detects all variables that have lost justification. The solver SMODELS for which the algorithm was originally written, sets all of them to false. In other words, it detects the greatest unfounded set and sets all the variables in it to false. The approaches in Section 3.4.3, on the other hand, use the idea of justification graph, but
aim at small and *loop-based* unfounded sets, and implement them in the context of SAT solving.

**Example 3.6.** Recall the program from Example 2.3 with founded variables $a, b, c$, standard variables $s, t$ and without the constraint in Example 2.3. Let us give each rule and constraint a name:

\[
\begin{align*}
R_1 &: a \leftarrow b \land \neg c \\
R_2 &: b \leftarrow a \land \neg c \\
R_3 &: c \leftarrow \neg a \land \neg b \\
R_4 &: b \leftarrow s \\
R_5 &: a \leftarrow \neg s \land t
\end{align*}
\]

Say the solver constructs an assignment $\theta$. As mentioned previously, we only mention source pointers, the edges from source pointers to positive variables are implicit.

Initially, the algorithm establishes the following source pointers: source($a$) = $R_5$, source($b$) = $R_4$ and source($c$) = $R_3$. Say the solver takes the decision: $\neg s \in \theta$. This affects $R_4$ which is the source of $b$, therefore, in the dejustification phase, source($b$) is temporarily removed. Since $b$ does not appear positively in any other current source, no other dejustification takes place. In the rejustification, the algorithm finds another source for $b$ which is $R_2$. Therefore, after the first decision, the source pointers are: source($a$) = $R_5$, source($b$) = $R_2$ and source($c$) = $R_3$.

Say the next decision is $\neg t \in \theta$. This affects source($a$) which is $R_5$. Therefore, in the dejustification phase, source($a$) is removed. Since $a$ appears positively in the source of $b$, source($b$) is also removed. There is no rejustification since there are no rules left for $a$ and $b$ that can be used as valid justifications that do not positively depend on $a$ or $b$. Therefore, $a$ and $b$ form an unfounded set, and are set to false, which propagates $c \in \theta$. Since all variables are fixed, $\theta$ is a complete assignment, and also a stable one, since there is no unfounded set w.r.t. $\theta$. Note that, if we did include the constraint in Example 2.3, then we would have failed since $\theta$ violates the constraint $s \lor t \lor b$.

### 3.4.2 ASP Via SAT + Loop Elimination

Recall from Section 2.4.5 that the stable models of a logic program are nothing but the propositional models of the completion of the program plus all its loop formulas. Also
recall that the number of loops formulas can be exponential in the number of variables in a logic program. The approaches presented by Lierler (2005), Lierler and Maratea (2004), Lin and Zhao (2004) build on this idea and work as follows. First, a SAT solver is run on only the completion of the program. If the program is unsatisfiable, then there are no stable models of the program. If the SAT solver returns a model, then there are two cases.

If the program is tight, then every propositional model of the completion is also a stable model, and we simply return the model. If the program is non-tight, then there is a possibility that the model is not a stable model. Therefore, it is checked whether there is some loop formula that is violated by the model. If yes, then that loop formula is added in the theory and the SAT solving step is repeated until either the theory becomes unsatisfiable or we find a loop-free model. If there are no loop formulas that are violated, the model is also a stable model.

Finding a violated loop formula from that a given model violates, if one exists, is trivial and one such approach is given in (Lin and Zhao, 2004). Given a model $M$, it computes $M^-$ which includes all variables that are in one or more unfounded set. $M^-$ is simply equal to all variables minus the positive variables in the least assignment of reduct of the program w.r.t. $M$. It then looks at the subgraph of the dependency graph that is induced by $M^-$, and looks for loops in each individual strongly connected component.

**Example 3.7.** Consider again the program in Example 2.3. The completion of the program is given in Example 2.5. Also, since $a$ and $b$ make up a non-trivial SCC in the positive dependency graph, the program is non-tight. Say we run a SAT solver on the completion, and the solver finds the solution $\theta_5$ of the model. The reduct w.r.t. $\theta_5$ includes the rules: $a \leftarrow b$ and $b \leftarrow a$, the least assignment sets both of these to false. Therefore, $\theta_5^-=\{a,b,c\} \setminus \emptyset$. The subgraph of the dependency graph induced by $\theta_5^-$ includes the loop that contains $a$ and $b$. Therefore, the algorithm above includes the loop formulas for this loop in the SAT theory (see Example 2.8) and runs the SAT solver again, which then finds one of the stable assignments $\theta_1$, $\theta_2$, or $\theta_3$ shown in Example 2.3. \qed
3.4.3 Unfounded Set Algorithms For Conflict-Driven Solvers

There are two algorithms that prune off unfounded sets w.r.t. partial assignments. These are the algorithms by Anger et al. (2006) and Gebser et al. (2012b). One property of these algorithms is that as opposed to the greatest unfounded set that DLV and SMODELS aim for, these algorithms aim to find small, loop-encompassing unfounded sets. These algorithms work well in tandem with conflict-drive SAT solvers that contain the completion of the rules. The small loops that these algorithms find, propagate and unit-resolve quite efficiently with these completion clauses, as well as previously learned loop formulas. We describe how these two algorithms can be adapted to work inside an explanation based lazy clause generation solver Chapter 5. Since we already describe these algorithms in that Chapter, we do not explain them here.

3.5 Solving Weight Constraint Rules In ASP

In Section 2.5.1, we described the weight constraint rules in ASP and their semantics. In this section, we discuss the state-of-the-art ASP implementation for weight constraint rules by Gebser et al. (2009a). We only restrict our attention to points that are directly relevant to this thesis and avoid describing the implementation in detail. Broadly, the implementation has two significant features: first, it follows lazy clause generation, and second, it extends the unfounded set algorithm used in CLASP to handle positive recursion among founded variables that occurs through weight constraints. The motivation behind using lazy clause generation for implementing weight constraints is the same as it is in finite domain solving, which is that the complete clausal representation is exponentially large, let us see how. Recall that the weight constraint rules can be written as:

\[ a \leftarrow w[b_1 = w_1, \ldots, b_j = w_j, \neg c_1 = w_{j+1}, \ldots, \neg c_k = w_{j+k}] \]

One way to encode the above weight constraint rule in normal rules is to introduce a founded variable \( b \) that becomes true whenever the sum above is greater than or equal to \( w \), and then change the above rule to: \( a \leftarrow b \). In order to assign \( b \) the desired value,
we need to post the following rules. Let \( B \) be the set of all weight literals above, i.e.,
\[
B = \{b_1 = w_1, \ldots, b_j = w_j, \neg c_1 = w_{j+1}, \ldots, \neg c_k = w_{j+k}\}.
\]
Let \( W \) be a function that maps a subset \( S \subseteq B \) of weight literals \( \{l_1 = w_1, \ldots, l_n = w_n\} \) to their net weight, i.e.,
\[
W(S) = w_1 + \ldots + w_n.
\]
Then, for each subset \( S \) of the literals, if \( W(S) \geq w \), then post the rule:
\[
b \leftarrow \bigwedge_{(i=w) \in S} l.
\]
Clearly, in general, the number of such rules required for the correct encoding is exponential in the size of the body, since it is in the order of number of subsets of \( B \). The algorithms described by Gebser et al. (2009a) lazily generates the clausal representation for above rules. The central idea of the approach is to keep the components that deal with pseudo-Boolean logic separate from the ones that deal with foundedness. The goal of one part of the algorithm is to keep the value of \( b \) consistent with weight literals while ignoring foundedness, while the second part is responsible for pruning off unfounded sets. The first one builds on the pseudo-Boolean constraints implementation in SAT solvers (Sheini and Sakallah, 2006). The second part involves extending the existing unfounded set algorithm of CLASP to detect loops that go through the rules above, without having explicit access to the rules of course. The crucial point that we want to get across in this subsection is that the solver does not ground, or transform, the weight constraint rules to normal rules, but has dedicated explanation-based propagators that reason directly with them.

### 3.6 Model Counting

We review two well known approaches for model counting. For a more detailed discussion of algorithms for model counting, see the work by Gomes et al. (2009).

#### 3.6.1 Solution Enumeration Using SAT Solvers

The first approach is simply to enumerate all solutions using the DPLL algorithm (Davis et al., 1962), but here we need to treat the backjumping case specially.

In traditional DPLL-algorithm (Davis et al., 1962), once a decision literal is retracted, it is guaranteed that all search space extending the current assignment has been exhausted.
Due to this, we can be certain that the search procedure is complete and does not miss any solution. This is not true, however, for modern SAT solvers (Moskewicz et al., 2001) that use random restarts and First-UIP backjumping. In the latter, the search backtracks to the last point in search where the learned clause is asserting, and that might mean backjumping over valid solution space. It is not trivial to infer from the current state of the solver which solutions have already been seen and therefore, to prevent the search from finding an already visited solution $\theta$, SAT solvers add the blocking clause $\neg \theta$ in the problem formulation as soon as $\theta$ is found, and they keep searching until they derive unsatisfiability.

3.6.2 DPLL-Style Model Counting

One of the most successful approaches for model counting extends the DPLL algorithm (see (Bacchus et al., 2009, Sang et al., 2004, Thurley, 2006)). Such model counters borrow many useful features from SAT solvers such as nogood learning, watched literals and backjumping etc to prune parts of search that have no solution. However, they have three additional important optimizations that make them more efficient at model counting as compared to solution enumeration using a SAT solver. A key property of all these optimizations is that their implementation relies on actively maintaining the residual program during the search. This requires visiting all clauses in the worst case at every node in the search tree.

Say we are solving $F$ and the current assignment is $\theta$. Let us describe the three optimizations.

The first optimization in model counting is cube detection; as soon as the residual is empty, we can stop the search and increment our model count by $\text{size}(\theta)$. This avoids continuing the search to visit all extensions of the cube since all of them are solutions of $F$.

**Example 3.8.** Consider the Boolean formula: $F_1 = \{ \neg b \lor a, \neg c \lor \neg a \lor b, \neg d \lor c, \neg e \lor c \}$. Suppose the current partial assignment is $\{ a, b, c \}$. The formula is already satisfied irrespective of values of $d$ and $e$. Instead of searching further and finding all 4 solutions, we can stop and record
that we have found a solution cube containing $2^2$ solutions, where 2 is the number of unfixed variables $(d, e)$.

We can generalize Example 3.8 and say that a cube has $2^k$ solutions, where $k$ is the number of unfixed variables.

The second optimization is caching (Bacchus et al., 2003) which reuses model counts of previously encountered sub-problems instead of solving them again as follows. Say we have computed the model count below $\theta_1$ and it is equal to $c$, we store $c$ against $F|_{\theta_1}$. If, later in the search, our assignment is $\theta_2$ and $F|_{\theta_2} = F|_{\theta_1}$, then we can simply increment our count by $c$ by looking up the residual.

Example 3.9. Refer to $F_1$ from Example 3.8. Suppose we first encountered the partial assignment $\theta_1 = \{d, c\}$. Then $F_1|_{\theta_1} = \{\neg b \lor a, \neg a \lor b\}$. After searching this subtree, we find that this subproblem has 2 solutions and cache this result. The subtree under $\theta_1$ thus has $2^{5-2-2} \times 2 = 4$ solutions. Suppose we later encounter $\theta_2 = \{\neg d, e, c\}$. We find that $F_1|_{\theta_2}$ is the same as $F_1|_{\theta_1}$. By looking it up in the cache, we can see that this subproblem has 2 solutions. Thus the subtree under $\theta_2$ has $2^{5-3-2} \times 2 = 2$ solutions.

The third optimization is dynamic decomposition and it relies on the following property of Boolean formulas: given a formula $G$, if (clauses of) $G$ can be split into $G_1, \ldots, G_n$ such that $\text{vars}(G_i) \cap \text{vars}(G_j) = \emptyset, 1 \leq i \neq j \leq n$, and $\bigcup_{i=1,n} \text{vars}(G_i) = \text{vars}(G)$, then $\text{count}(G) = \text{count}(G_1) \times \ldots \times \text{count}(G_n)$. Model counters use this property and split the residual into disjoint components and count the models of each component and multiply them to get the count of the residual. Furthermore, when used with caching, the count of each component is stored against it so that if a component appears again in the search, then we can retrieve its count instead of computing it again.

Example 3.10. Consider $F_2 = \{a \lor \neg b \lor c, c \lor \neg d \lor e, e \lor f\}$ and a partial assignment $\{\neg c\}$. The residual program can be decomposed into two components $\{a \lor \neg b\}$ and $\{\neg d \lor e, e \lor f\}$ with variables $\{a, b\}$ and $\{\neg d \lor e, e \lor f\}$ respectively. Their counts are 3 and 5 respectively, therefore, the number of solutions for $F_2$ that extend the assignment $\{\neg c\}$ is $3 \times 5 = 15$.
3.7 Compiling d-DNNF Via Model Counting

In this section, we briefly review the algorithm described by Muise et al. (2012) for compiling d-DNNF by essentially recording the trace of a DPLL-model counter equipped with cube detection and component caching such as SHARPSAT (Thurley, 2006).

The basic idea is that every sub-problem (residual) $F$ encountered during the search is represented by a node in the d-DNNF, let us call it $\text{node}(F)$. The root node represents the CNF $C$ formula that the model counter solves, i.e. $\text{node}(C)$. Next, we define the operations that the solver performs on a formula $F$. If $F$ is equal to the literal $l$, then $\text{node}(F) = l$. If the solver propagates literals $l_1, \ldots, l_n$, then $\text{node}(F) = l_1 \land \ldots \land l_n \land \text{node}(F|_{l_1,\ldots,l_n})$. If the solver finds that $F$ is equal to some previously recorded formula $F'$, then $\text{node}(F) = \text{node}(F')$. If the solver finds that $F$ can be decomposed into the disjoint components $F_1, \ldots, F_n$, then $\text{node}(F) = \text{node}(F_1) \land \ldots \land \text{node}(F_n)$. If a solver takes a decision $l$, then the current node is replaced with $\text{node}(F|_l) \lor \text{node}(F|_{\neg l})$. Finally, if the solver fails, then $\text{node}(F) = \text{false}$. In the end, the d-DNNF is simplified to remove nodes that evaluate to false (either false leaf nodes, or conjunctions that have at least one false child).

**Example 3.11.** Refer to the d-DNNF in Example 2.12. It can be seen as a trace of the formula $F$ given in the same example. The first OR node is the result of the decision $b$. The left child represents the sub-problem $F|_b$ and the right child of the OR node represents the sub-problem $F|_{\neg b}$. $F|_b$ decomposes into disjoint sub-problems $(-a \lor c)$ and $(d \lor -e)$ whose nodes are connected with an AND node. Next, while solving the first component, when the decision $a$ is taken, the solver unit propagates $c$ and so on. There are no failures in the example, so no simplification is required, and we get the d-DNNF as shown in Example 2.12.
Part I

Combining Constraint Programming and Answer Set Programming
Chapter 4
Motivation

In this chapter, we motivate combining ASP and CP. This chapter lays the foundation for the rest of the chapters in this part of the thesis.

We begin by discussion of foundedness for Boolean variables, which is a strength of ASP systems, and absence of which is a weakness for CP systems. Then we look at the standard grounding bottleneck, which is caused by the inability of ASP systems to succinctly represent integer variables. This provides the motivation for the next chapter, which describes Constraint Answer Set Programming (CASP), a useful formalism that does not suffer from standard grounding bottleneck, and also supports foundedness over Boolean variables.

After discussion of standard grounding bottleneck, we discuss another type of grounding bottleneck that arises due to absence of algorithms to deal with foundedness over bounds of numeric variables. Founded grounding bottleneck provides the motivation for our formalism called Bound Founded Answer Set Programming (BFASP) which extends stable model semantics over numeric bounds. We present the theory of BFASP, an algorithm for its implementation, and its various language aspects in Chapters 6, 7 and 8 respectively.

4.1 Strengths Of ASP

In Chapter 3, we saw how modern ASP solving (Gebser et al., 2007, 2012b) builds on propositional satisfiability (SAT) solving (Mitchell, 2005). Recall that ASP solvers have a competitive edge over SAT solvers in problems whose model involves some notion of transitive closure, e.g., reachability or connectivity in a graph, or more generally, some kind of inductive definition with one or more base cases, and one or more inductive cases. These problems are much more naturally modeled in ASP as compared to SAT. This is due to the difference in semantics of both systems; ASP solvers implement stable model semantics (Gelfond and Lifschitz, 1988a) which minimizes the number of variables that
are true in a given logic program while a SAT solver only looks for an assignment that satisfies all the given clauses. In ASP, in order for a variable to be true, it must have some rule as a support that justifies it being true. Furthermore, no set of variables can circularly support one another. E.g. given two rules \( a \leftarrow b \) and \( b \leftarrow a \), the only valid solution in stable model semantics is where \( a \) and \( b \) are both false, whereas in propositional semantics, both the variables being true is also a valid solution.

We demonstrate the central points about strengths and weaknesses of different systems with the help of a problem called the Minimum Connected Dominating Set (MCDS) problem whose description is as follows. A dominating set is a set of nodes such that every node in the graph is either in the set or has at least one neighbour in the set. The objective is to find a dominating set of minimum cardinality such that the subgraph induced by dominating nodes is connected. Connected dominating sets are very useful, especially in computer and mobile networks, as we can direct all communication through them. We can be sure that as long as the nodes in this set are online, we can deliver message from any source to any destination. For this reason, it is also desirable to have such a set with minimum cardinality, and hence the problem name.

Let us now look at the ASP encoding of the problem given in Figure 4.1.\(^8\) Let us say that a vertex \( X \) is given in the input as \( \text{vtx}(X) \), an edge from node \( X \) to node \( Y \) as \( \text{edge}(X,Y) \). Let us assume that the edge relation is symmetric, i.e. \( \text{edge}(X,Y) \rightarrow \text{edge}(Y,X) \).

\( R_1 \) introduces a decision variable \( \text{dom} \) for each vertex specifying whether it is a dom-

---

\(^8\)Based on the model from Potassco group in the second ASP competition: [http://dtai.cs.kuleuven.be/events/ASP-competition/encodings.shtml](http://dtai.cs.kuleuven.be/events/ASP-competition/encodings.shtml)
4.1 Strengths Of ASP

\[
\begin{align*}
R_1 & \{ \text{dom}(U) : \text{vtx}(U) \}. \\
R_2 & \text{in}(V) \leftarrow \text{dom}(V). \\
R_3 & \text{in}(V) \leftarrow \text{edge}(U, V) \land \text{dom}(U). \\
C_1 & \leftarrow \text{vtx}(U) \land \neg \text{in}(U). \\
R_4 & \text{reach}(U) \leftarrow \text{dom}(U) \land \neg \exists V < U : \text{dom}(V). \\
R_5 & \text{reach}(V) \leftarrow \text{reach}(U) \land \text{dom}(V) \land \text{edge}(U, V). \\
C_2 & \leftarrow \text{dom}(U) \land \neg \text{reach}(U). \\
O & \text{minimize}\{ \text{dom}(U) : \text{vtx}(U) \}. 
\end{align*}
\]

Figure 4.2: ASP encoding of Minimum Connected Dominating Set (MCDS)

ating vertex or not. \(R_1\) is a choice rule which says that for a given node \(U\), this rule can be a justification for \(\text{dom}(U)\) if it is true (essentially making \(\text{dom}(U)\) a standard variable). \(R_2\), \(R_3\) and \(C_1\) model that every node must either be a dominating node or have a neighbour that is dominating. This is done with the help of an auxiliary predicate \(\text{in}\) that becomes true when at least one of the conditions is met. \(C_1\) says that there can be no node for which \(\text{in}\) is false. The objective, given by \(O\), is to minimize the cardinality of the dominating set.

Before we do so, let us make an observation.

The program so far does not involve any positive recursion, recall from Chapter 2 that such a program is tight and in this case, the classical or propositional models of its completion coincide with the stable models of the program. But once we add the connectivity constraint, which we do next, that is no longer true.

Figure 4.2 gives the complete ASP encoding of MCDS with the addition of \(R_4\), \(R_5\) and \(C_2\) in Figure 4.1. \(R_4\) and \(R_5\) define the predicate \(\text{reach}\) that is used to model the connectivity constraint of the induced subgraph by the dominating nodes. \(R_4\) encodes the base case for reachability, specifying that the node with the lowest index is reachable by definition. This choice is arbitrary and its purpose can be satisfied by any criterion to select a dominating node. \(R_5\) is a recursive case for reachability and it says that a dominating neighbour of a reachable node is also reachable. The constraint \(C_2\) says that
all dominating nodes must be reachable.

The program is no longer tight, and this is where it becomes inconvenient (and inefficient w.r.t. the solving algorithms) to model this in propositional logic. The completion of the above program is obviously not sufficient since we can simply set all \( \text{reach} \) variables to true, without making any \( \text{dom} \) variable true, and it would still be a propositional model of the completion.

An important point to note here is that a choice rule \( \{a\} \) in ASP basically removes the foundedness condition from the variable \( a \); it specifies that \( a \) does not need any justification to be true. Equivalently, if \( a \) is true, then this choice rule can act as a justification for it. Unlike other variables in an ASP program, a variable appearing in a choice rule does not have to be equal to false if there is no other rule that makes it true. The IDP system (Wittocx et al., 2008) clearly distinguishes between open and defined variables. By default, in an ASP program, a variable is defined, pointing to the fact that its truth value needs to be founded by some rule. Once it appears in a choice rule, however, it becomes an open, or a standard variable, meaning the solver is free to assign it any value as long as it satisfies all the constraints.

### 4.1.1 Modelling Boolean Foundedness In SAT

In this subsection, we review several approaches that encode foundedness over Boolean variables in SAT. We point out that these translation based approaches do not scale as well as ASP solvers on the original logic programs, and therefore, dedicated algorithms to implement foundedness over Boolean variables is a desirable feature in a combinatorial solver. Let us look at some of the translation based proposals now.

- We could construct the equivalent propositional theory of a logic program by taking its completion and adding all loop formulas to it and then solving that theory to get the stable models of the program. As discussed in the Chapter 2, this could make the encoding exponential in the number of nodes. Another approach that also produces an exponentially larger SAT theory, is given by Vlasselaer et al. (2014).

- The approach given by Janhunen (2004) produces a SAT encoding that is loglinear
in the size of a logic program. First, a \textit{level rank} is associated with each founded variable. Second, if a rule is used to derive the head (which is modelled by a separate set of variables), then the level ranks of all positive variables in the rule body must be strictly less than the level rank of the head. The encoding ensures that we never get two or more founded variables that derive each other in a circular way, since that gives rise to an infeasible order on level rank variables. E.g. if we have a current assignment which makes all rule bodies for \textit{a} false except \textit{a} $\leftarrow \textit{b}$, and \textit{a} is non-false, then the encoding ensures that the level rank of \textit{b} is less than the level rank of \textit{a}. Similarly, if all rule bodies for \textit{b} are false except \textit{b} $\leftarrow \textit{a}$, and \textit{b} is non-false, then the encoding ensures that the level rank of \textit{a} is less than the level rank of \textit{b}. The two constraints are impossible to satisfy, therefore, \textit{a,b} cannot be \textit{true} in any extension of the current assignment. The logarithmic size comes from the binary encoding of level ranks and constraints associated with them. Although the size of the translated SAT program is small, the performance is not as good as conflict-driven answer set solving, as shown in the experiments by Gebser et al. (2012b).

Another way to use the same intuition behind level ranks, i.e. inducing order on founded variables such that unfounded sets of variables imply an infeasible order, is to use a quadratic number of variables instead of the binary encoding. Let us demonstrate that by modelling the connectivity condition in MCDS. Let \textit{n} be the number of nodes in the graph:

\begin{align*}
R_4 & \quad \text{reach}(U,1) \leftrightarrow \text{dom}(U) \land \forall V : \text{dom}(V).
R_5 & \quad \text{reach}(U,T) \leftrightarrow \text{reach}(U,T-1).
R_6 & \quad \text{reach}(V,T) \leftrightarrow \text{reach}(U,T-1) \land \text{dom}(V) \land \text{edge}(U,V).
C_2 & \quad \neg \text{dom}(U) \lor \text{reach}(U,n).
\end{align*}

First of all, we assume that the all previous rules and constraints have been translated appropriately in propositional logic, using e.g. the Clark completion. \textit{R}_4, \textit{R}_5, \textit{R}_6 and \textit{C}_2 work as the propositional replacement for encoding the connectedness
condition previously done with the help of ASP rules. Furthermore, we use $\leftrightarrow$ for better readability, although these should be translated to propositional clauses. The encoding ensures non-circular derivation of the reach variable, by introducing a time parameter whose domain varies from 1 \ldots n, which is the reason it uses quadratic number of variables. The idea is that the node with the smallest index is reachable at time step 1 by definition ($R_4$); that node itself ($R_5$) as well as its immediate neighbours that are in the dominating set are reachable in the time step 2 ($R_6$), and so on. Note that we have omitted the time domain from rule bodies for brevity $(vtx(T) \land vtx(T - 1))$. Finally, clause $C_2$ says that every node in the dominating set must be reachable in the last time step. The program is tight since we have broken recursion by using a quadratic number of variables, and we can use a SAT solver on the program to give us the correct answer (see Section 2.4.4).

### 4.2 Standard Grounding Bottleneck

Let us turn our attention towards the problem of standard grounding bottleneck for ASP and SAT that arises due to their inability to succinctly represent numeric variables. To motivate this, let us change the MCDS problem. First, assume that the graph is weighted now, $\text{edge}(U, V, W)$ means that the edge from $U$ to $V$ has weight $W$, and the weights represent the widths of edges. Second, we want that the dominating subgraph must have a uniform width that is at least equal to $\text{min}_\text{width}$, i.e., all edges between dominating nodes must have a fixed width which is greater than or equal to $\text{min}_\text{width}$. To achieve this, we are allowed a certain number of total repairs $\text{max}_\text{rep}$, i.e. additions and subtractions to the existing edges. Let us call this problem MCDS with uniform width, or MCDS-UW.

Figure 4.3 gives the ASP model for MCDS-UW. $R_6$ declares the predicate $\text{num}$ defined over an integer domain $l \ldots u$, where $l$ is a sufficiently small integer and $u$ is a sufficiently large integer. \(^9\) Once grounded, the number of propositional variables used to represent this integer domain in the ground program is equal to $u - l + 1$. In $R_7$, we decide the width of the dominating subgraph; we achieve this by saying that $\text{width}(W)$ holds for

\(^9\)Note that upper bounds from cardinality constraint rules can be removed. A rule $l \{L\} u$ where $L$ is a set of literals can be replaced with $l \{L\}$ and $\leftarrow u + 1 \{L\}$. 
4.2 Standard Grounding Bottleneck

\[ R_6 \quad \text{num}(l \ldots u). \]

\[ R_7 \quad \{ \text{width}(W) : \text{num}(W) \land W \geq \text{min}_{width} \} 1. \]

\[ R_8 \quad \{ r(U, V, R) : \text{num}(R) \land \text{edge}(U, V, W) \} 1. \]

\[ C_3 \leftarrow \text{edge}(U, V, W) \land \text{not dom}(U) \land \text{not } r(U, V, 0). \]

\[ C_4 \leftarrow r(U, V, R) \land \text{edge}(U, V, W) \land \text{dom}(U; V) \land \text{width}(W_1) \land W \neq W_1 + R. \]

\[ C_5 \leftarrow \text{max}_{rep} + 1 \{ r(U, V, R) : \text{edge}(U, V, W) \land \text{num}(R) = |R| \}. \]

Figure 4.3: ASP encoding of uniform width constraints for MCDS with uniform width (MCDS-UW)

precisely one value of \( W \) that is a number greater than or equal to \( \text{min}_{width} \). Similarly, \( R_8 \) assigns a unique repair value \( r(U, V, R) \) to the edge between \( U \) and \( V \). \( C_3 \) says that any edge outside the dominating subgraph should not be repaired (the repair value equals to 0). This constraint is not strictly needed for correctness, but there is no good reason to repair edges that are not in the dominating subgraph. \( C_4 \) says that for any edge within the dominating subgraph, the original width plus the repair must be exactly equal to the width that we choose in \( R_7 \). The last constraint \( C_5 \) says that the total number of repairs cannot exceed \( \text{max}_{rep} \).

The important point in the encoding of MCDS-UW is that what should ideally be one variable with a large integer domain, is represented by a set of propositional variables whose cardinality is equal to the domain. Furthermore, notice that for a given edge, ideally, we should have only one ground constraint instance from \( C_4 \) that says that the resulting width after the repair should be correct. Instead, the number of rules for one edge (fixed values of \( U \) and \( V \), say \( u \) and \( v \) respectively), is linear in the size of ground instances of \( r(u, v, R) \). Although, in this example, we only get a linear number of ground constraints, if a constraint has two or more of such variables, then as already explained in great detail in Example 3.4, the number of constraints can be prohibitive since we need to consider every combination of values in the domains of variables in the worst case, and have a corresponding ground instantiation for every value. This blows up the problem size, and incurs the standard grounding bottleneck.

In the next chapter, we describe Constraint ASP, a formalism that includes both founded
and standard Booleans, ASP rules, numeric variables, and generic constraints over variables. The implementation of CASP removes standard grounding bottleneck and also allows efficient solving of Boolean foundedness.

4.3 Founded Grounding Bottleneck

We just saw how since ASP systems such as SMODELS and CLASP only deal with Boolean variables, they are inefficient for solving problems that are naturally modelled with integers, especially if they have large domains. These combinatorial problems (e.g. scheduling) are ubiquitous in Computer Science, which makes ASP a poor choice to model and solve them. Constraint Programming solvers and Mixed Integer Programming solvers, on the other hand, are excellent candidates for these problems as they support numeric variables natively.

We also saw that constraint solvers, unfortunately, suffer from the same inefficiency as SAT solvers regarding problems like reachability that require foundedness. A hybrid system like CASP that has the best of both worlds, i.e., foundedness for Boolean variables from ASP, and native support for integers and constraints over them from CP, address both these concerns. However, we now propose that even this hybrid approach is not sufficient, and there exists a type of grounding bottleneck that is still not removed by combining the strengths of CP and ASP solvers. Let us motivate this by modifying MCDS once again.

Let us say that the edges have weights, just like they did for MCDS-UW, but now, the weights represent physical distances between the given nodes. Moreover, we have the constraint that the diameter, which is the maximum distance between any two nodes of the dominating subgraph is less than a certain given value $k$. Let us call this problem MCDS with Bounded Diameter or MCDS-BD. This problem has applications in computer networks (Buchanan et al., 2014, Kim et al., 2009). Let $d(X,Y)$ represent the distance (shortest path) between two dominating nodes $X$ and $Y$. In MCDS, it is sufficient to check for reachability of every dominating node from an arbitrary node to ensure connectedness. However, to enforce the new constraint, we need a distance variable for each pair of
nodes in the dominating set. We can replace $R_4$, $R_5$, and $C_2$ in the ASP encoding of MCDS in Figure 4.2 with the following:

$$R_4 \quad d(U, U) \leq 0 \leftarrow \text{dom}(U).$$

$$R_5 \quad d(U, T) \leq d(V, T) + W \leftarrow \text{dom}(T) \land \text{dom}(U) \land \text{dom}(V) \land \text{edge}(U, V, W).$$

$$C_2 \leftarrow \text{dom}(U) \land \text{dom}(V) \land d(U, V) > k.$$  

$R_4$ is the base case for $d$ and it says that the distance from a dominating node to itself is at most 0. $R_5$ is a recursive rule that specifies that for two dominating neighbours and a dominating node $T$, the distance between one end of the node to $T$ is at most the distance between the other end and $T$, plus the weight of the edge. Finally, the constraint $C_2$ establishes that the distance between any two dominating nodes must be at most $k$. It is unnecessary to include the previous reachability rules since finite distances between all pairs of dominating nodes implies that the dominating set is connected.

Rules like $R_4$ and $R_5$ on integer variables are clearly not supported by current ASP systems. The semantics that we wish to associate with the distance variables is that firstly, if there are no rules supporting them, then they are equal to $\infty$. Secondly, any rule for a distance variable justifies a value lower than $\infty$ and thirdly, the upper bounds of these variables cannot form a circular justification. E.g. if there are two rules: $a \leq b$ and $b \leq a$, then any solution where $a$ and $b$ are equal to a finite value should be rejected, and the only stable solution should be one where both are equal to $\infty$. The distance variable is essentially an upper-bound founded (ub-founded) variable, for which the upper bound needs to be justified by some rule.

Figure 4.4 shows how we can encode these upper-bound founded distance variables in ASP along with our desired semantics by replacing $R_4$, $R_5$, and $C_2$. In the encoding, $m$ is a sufficiently large integer and $d_{ub}(U, V, N)$ specifies that the distance between the dominating nodes $U$ and $V$ is at most $N$ (the subscript $ub$ stands for upper-bound). $d(U, V, D)$ is defined as the minimum value for which $d_{ub}(U, V, D)$ is true. Unfortunately, an ASP solver on this encoding quickly runs into the grounding bottleneck problem as we
Motivation

\[ d_{ub}(U, V, N) \leftarrow \text{dom}(U) \land \text{dom}(V) \land d_{ub}(U, V, N - 1), N < m. \]
\[ d_{ub}(U, U, 0) \leftarrow \text{dom}(U). \]
\[ d_{ub}(U, T, D + W) \leftarrow \text{dom}(T) \land \text{dom}(U) \land \text{dom}(V) \land \text{edge}(U, V, W) \land d(V, T, D). \]
\[ d(U, V, D) \leftarrow \text{dom}(U) \land \text{dom}(V) \land d_{ub}(U, V, D) \land -d_{ub}(U, V, D - 1). \]
\[ \leftarrow \text{dom}(U) \land \text{dom}(V) \land d(U, V, D) \land D > k. \]

Figure 4.4: ASP encoding of diameter constraint in MCDS with bounded diameter (MCDS-BD)

increase edge weights and the bound on diameter. This is the motivation behind Bound Founded Answer Set Programming (BFASP), i.e., to support founded numeric variables and rules like \( R_4 \) and \( R_5 \) without grounding them.

The symmetric analog for a ub-founded variable is a lower-bound founded (lb-founded) variable, which is by default equal to \(-\infty\) (false for Boolean) and further rules for it can justify a greater value on its lower bound. In this generalization, ASP variables are simply lb-founded Boolean variables. For simplicity, we only consider lb-founded variables, and refer to them as founded variables. This simplification is possible because we can replace all ub-founded variables, their rules, and their constraints by corresponding lb-founded variables with similar rules and constraints. E.g., for the above problem, let \( d(U, V) \) represent the negative of the distance between \( U \) and \( V \), then we can perform this transformation as follows:

\[
R_4 \quad d(U, U) \geq 0 \leftarrow \text{dom}(U).
\]
\[
R_5 \quad d(U, T) \geq d(V, T) - W \leftarrow \text{dom}(T) \land \text{dom}(U) \land \text{dom}(V) \land \text{edge}(U, V, W).
\]
\[
C_2 \quad \leftarrow \text{dom}(U) \land \text{dom}(V) \land d(U, V) < -k.
\]

Since there are no rules and founded variables in CP and MIP solvers, MCDS with bounded diameter as defined above cannot be efficiently solved by them. The MIP formulation as given by Buchanan et al. (2014) encodes each distance variable with \( K \) propo-
sitional variables, meaning that the problem size increases with $K$. Our encoding above that uses founded numeric variables does not suffer from this problem.

In Chapter 6, we present Bound Founded ASP. In BFASP, we have founded Booleans, standard Booleans and numeric, as well as founded numeric variables. The set of rule forms that BFASP allows is generic enough to incorporate the shortest path/distance rules that we saw above. In Chapter 7, we describe an algorithm that implements BFASP semantics, and avoids the founded grounding bottleneck.

## 4.4 Grounding Bottleneck And Weight Constraint Rules

It is important to distinguish the two types of grounding bottlenecks described in this chapter, standard and founded, from the grounding bottleneck that arises due to absence of support for weight constraint rules (which are described in Sections 2.5.1 and 3.5). The latter type of grounding bottleneck, as opposed to the two types shown in this chapter, does not arise due to the inability to reason with integers natively. Here, it is also important to point to the distinction between weight constraint rules and pseudo-Boolean constraints. Pseudo-Boolean constraints are a special case of weight constraint rules. As long as there is no positive recursion occurring through weight constraint rules in a problem, pseudo-Boolean constraints are sufficient.

CASP semantics, as described by Gebser et al. (2009c), does not support weight constraint rules. The extension of semantics to allow weight constraints is certainly possible, although we do not discuss it in the thesis since it is not directly relevant. The CASP semantics is, however, well-defined for pseudo-Boolean constraints.

Major CASP implementations support reasoning with pseudo-Boolean constraints, although they may or may not support weight constraint rules. E.g. the implementation described in the next chapter only adds a normal rule form in a constraint solver, therefore, it does not allow positive recursion in founded variables through weight constraint rules. On the other hand, CASP implementations that use some ASP solver that already supports weight constraint rules in its architecture, like CLINGCON, do support weight
constraint rules.\textsuperscript{10}

Our proposed system, BFASP, supports a rich set of rule forms. Both the theory given in Chapter 6 as well as the implementation given in Chapter 7, are sufficient to deal directly with weight constraint rules, without transforming them to any other rule form.

\textsuperscript{10}Technically, these are implementations of the extended CASP that incorporates the semantics for weight constraint rules.
Chapter 5
Constraint Answer Set Programming

In Chapter 4, we looked at how standard grounding bottleneck arises when we try to model numeric variables with large domains in ASP. On the other hand, Constraint Programming natively deals with numeric variables. In this Chapter, we look at Constraint Answer Set Programming (CASP) (Gebser et al., 2009c), which is a hybrid ASP/CP system, and effectively removes the standard grounding bottleneck.

We begin this chapter with a formal discussion about the semantics of CASP. Next, we describe an implementation of CASP in the lazy clause generation solver CHUFFED (Chu, 2011). Our approach uses two algorithms for unfounded set detection, that were originally devised for efficient ASP solving, and implements them as propagators in CHUFFED. To the best of our knowledge, all previous efforts in implementing CASP have focused on either extending an ASP solver with CP capabilities, or combining off-the-shelf ASP and CP solvers. This is the first work that shifts that view, and proposes CASP solving as a natural extension of CP solving based on lazy clause generation. Finally, we review some related work, carried out in the ASP research community, that aims to remove standard grounding bottleneck.

Throughout this chapter, as we reiterate shortly, whenever we refer to founded variables, we refer to Boolean founded variables, i.e. $V_F \subseteq V_B$.

5.1 Constraint Answer Set Programming

A Constraint Answer Set Program (CASP) is a tuple $(V, R, C, D)$ where $V$ is a set of standard and founded variables with the initial domain $D$, the standard variables could be either numeric or Boolean, but the founded variables are strictly Boolean, i.e. $V_F \subseteq V_B$; $C$ is a set of constraints over $V$, and $R$ is a set of rules of the form: $a \leftarrow b_1 \land \ldots \land b_n \land \neg c_1 \land \ldots \land \neg c_m$ where $a \in V_F$ and the variables of literals in the body could be standard
or founded. Note that we have only extended the ASP formalism, as defined in Section 2.4.1 to include numeric variables, and more general constraint types.

The stable model semantics for CASP is defined similar to how it is defined in 2.4.1, but we reproduce it here. Given an assignment $\theta$ and a program $P$, $P^\theta$ is a set of positive rules that is obtained as follows: for every rule $r$, if any $\theta(c_i) = \text{true}$, or $\theta(b_j) = \text{false}$ for any standard positive literal, then $r$ is discarded, otherwise, all negative literals and standard variables are removed from $r$ and it is included in the reduct. Let us write the reduced rule as $r^\theta$. An assignment $\theta$ is a stable assignment of a program $P$ iff it satisfies all its constraints and $\theta =_{\nu_f} \text{Least}(P^\theta)$.

The definition of unfounded sets for CASP remains the same as the one given in Section 2.4.5. The completion clauses are also similarly obtained as the ones defined for ASP programs in Section 2.4.3.

**Example 5.1.** Consider the following CASP with founded Booleans $a, b, c$, standard Booleans $s, t$, standard integers $x_1, x_2$ both with initial domains $\{5 \ldots 10\}$, and the following rules and constraints:

$$
\begin{align*}
R_1 &: a \leftarrow b \land \neg c & R_2 &: b \leftarrow a \land \neg c & R_3 &: c \leftarrow \neg a \land \neg b & R_4 &: b \leftarrow s \\
R_5 &: a \leftarrow \neg s \land t & C_1 &: t \leftrightarrow \text{alldifferent}(x_1, x_2) & C_2 &: s \leftrightarrow x_1 \geq 6
\end{align*}
$$

Now consider the following assignments:

$$
\begin{align*}
\theta_1 &= \{ a \mapsto \text{true}, \ b \mapsto \text{true}, \ c \mapsto \text{false}, \ s \mapsto \text{true}, \ t \mapsto \text{false}, \ x_1 \mapsto 7, \ x_2 \mapsto 7 \} \\
\theta_2 &= \{ a \mapsto \text{false}, \ b \mapsto \text{false}, \ c \mapsto \text{true}, \ s \mapsto \text{false}, \ t \mapsto \text{false}, \ x_1 \mapsto 5, \ x_2 \mapsto 5 \} \\
\theta_3 &= \{ a \mapsto \text{true}, \ b \mapsto \text{true}, \ c \mapsto \text{false}, \ s \mapsto \text{false}, \ t \mapsto \text{false}, \ x_1 \mapsto 5, \ x_2 \mapsto 5 \} \\
\theta_4 &= \{ a \mapsto \text{true}, \ b \mapsto \text{true}, \ c \mapsto \text{false}, \ s \mapsto \text{true}, \ t \mapsto \text{false}, \ x_1 \mapsto 7, \ x_2 \mapsto 8 \}
\end{align*}
$$

$\theta_1$ satisfies all constraints. $P^\theta$ includes the following reduced rules:
\[ R^\theta_1 : a \leftarrow b \quad R^\theta_2 : b \leftarrow a \quad R^\theta_4 : b \]

\[ \text{Least}(P^\theta_1) = \{a \mapsto \text{true}, b \mapsto \text{true}, c \mapsto \text{false}\} \]

which is equal to \( \theta_1 \) on founded variables, and hence \( \theta_1 \) is a stable assignment.

Similarly, \( \theta_2 \) is also a stable model.

Although \( \theta_3 \) satisfies all constraints, the reduct is:

\[ R^\theta_3 : a \leftarrow b \quad R^\theta_3 : b \leftarrow a \]

The least assignment sets \( a, b, c \) to false, which is not equal to \( \theta_3 \) on founded variables. Therefore, \( \theta_3 \) is not a stable assignment.

\( \theta_4 \) does not satisfy the constraint \( C_1 \) since \( x_1 \) and \( x_2 \) have different values, yet \( t \) is equal to false. Therefore, \( \theta_4 \) is not a stable assignment.

Although, in the definition of CASP semantics above, we only mention numeric or Boolean variables, the semantics does not need any modification to be defined for any type of standard variable (e.g. set variables) and constraints associated with it. We restrict to numeric and Boolean variables in this thesis for simplicity.

Another important point is that although the semantics above is defined for numeric variables, in practice, it is possible that a CASP implementation only supports a subset of variables. For example, CASP implementations that are based on constraint solvers, such as the one by Gebser et al. (2009c) and the one presented in this chapter, only support finite domain integer variables, and not continuous ones. It is important to realize that this is purely a limitation of the implementation, and not of CASP theory.

### 5.1.1 Comparison With CASP Semantics By Gebser et al. (2009c)

The semantics for CASP defined above is similar to the original one given by Gebser et al. (2009c), the differences are only superficial, so let us address them here. First of all, the
constraint atoms of Gebser et al. (2009c) are the same as standard Booleans. Secondly, our reduct, as defined above, removes standard as well as negative literals from the rules, and eliminates a rule if any one of these literals is violated, in one pass, whereas the semantics of Gebser et al. (2009c) achieves this in two steps. First, the constraint reduct performs this filtering considering constraint atoms only, and then the regular GL-reduct of Gelfond and Lifschitz (1988b) filters out literals and rules based on negative literals. It is merely a different point of view, and makes no difference in the semantics, as the resulting set of rules in the reduct is the same. Thirdly, if a constraint \( c \) in our set of constraints \( C \) is not reified (meaning true \( ⇔ \) \( c \)), then in order to enforce this constraint using semantics of Gebser et al. (2009c), we must introduce a constraint atom \( x \), reify it with \( c \), that is, post \( x ⇔ c \) and then include an integrity constraint \( \leftarrow ¬x \) in the constraint logic program.

5.2 Implementing CASP In A Lazy Clause Generator

In this section, we present an approach to implement CASP in the lazy clause generation solver CHUFFED. Before we describe our implementation in detail, let us describe briefly how propagation works in CHUFFED. A propagator can subscribe to an event \( e \), written \( \text{Subscribe}(e) \). When the event \( e \) takes place, the \( \text{WakeUp} \) function of the propagator is called. At this point, the propagator can queue up for propagation, using the function \( \text{Queue} \). Since a single event can wake up more than one propagator, each propagator has a priority; any woken propagators are added to the queue in priority order. The \( \text{Propagate} \) function of a propagator is called after all the higher priority propagators have finished. The code of the \( \text{Propagate} \) function can choose to requeue itself, using the function \( \text{Requeue} \), after it has done some work, if it wants higher priority propagators to run before it does some more work.

For unfounded set calculation, we have implemented two approaches taken from existing literature. The first one is based on the approach outlined by Gebser et al. (2007), which is the combination of SMODELS' source pointer technique (Section 3.4.1) with the unfounded set computation algorithm described by Anger et al. (2006). The second approach follows Gebser et al. (2012b) in combining the source pointer technique with a
5.3 Initial Calculations

The initial calculations, described in this section, are the same for both approaches. Let \( \text{Comp}(P) \) represent the completion clauses of \( P \) as described in Section 2.4.3. Let \( \text{bodies}(P) \) represent the set of body variables that are introduced to represent bodies during completion, again as described in Section 2.4.3. Let \( \text{posInRules}(a) \) give the set of rules in which the founded variable \( a \) positively appears. In this chapter, for a rule \( r \), whenever we write \( \text{body}(r) \), instead of referring to the conjunctive body, we actually refer to the auxiliary variable that is introduced in the completion to represent the body. Therefore, in this chapter, \( \text{body}(r) \) is an actual standard variable. We slightly abuse the notation for different unfounded set computation algorithm (described in that paper). We call our implementation of the first approach \textsc{ANGER}, and the second one \textsc{GEBSER}, after the authors of their unfounded set algorithms.

Computing unfounded sets is inherently more expensive than most other propagators. We therefore want to invoke the unfounded set propagator as rarely as possible, which requires its priority to be low. This low priority is required for another reason as well: the algorithms used by the unfounded set propagator need to run after unit propagation has finished, so that they have access to a consistent valuation of all the Boolean solver variables. If they do not, then the work that they do is likely to turn out to be wasted.

In the rest of this section, we describe the two approaches in detail. We provide the pseudo-code using the notation used in this thesis. The paper describing the first approach (Anger et al., 2006) has some missing pseudo-code about how the unfounded set algorithm fits in a conflict-driven solver, although the paper does provide some of these details in prose. We provide all those details for the sake of completeness. There are no substantial differences in the second algorithm as compared to the one by Gebser et al. (2012b), except for the notation.

For all the algorithms used in this section, we assume that they access the domain \( D \), and all higher priority propagators have run.
better readability, and use $\theta(r)$ to refer to $\theta(body(r))$. Furthermore, when we say that $r$ is non-false, we mean that $\theta(body(r)) \neq false$. When the solver is initialized, prior to any propagation and search, we calculate $Comp(P)$ and record its clauses.

We then calculate the strongly connected components of the positive dependency graph of the given CASP program. We number each component, and assign each founded variable and body the number of the component it is in. The only founded variables of interest are those whose components contain more than one founded variable, since only they can ever participate in a non-trivial, loop-based unfounded set. The calculation of the SCCs allows us to distinguish between these cyclic founded variables, and all other variables, which are acyclic.

### 5.3.1 Establishing Source Pointers

Both our unfounded set detection algorithms are based on the idea of source pointers. As already described in detail in Section 3.4, each cyclic founded variable has a source, which is a non-false rule $r$ such that founded variables in $r^+$ are not unfounded.\(^{11}\) As long as the source of an founded variable is non-false, the variable has evidence of not being unfounded. If the source of a variable becomes false, then we must look for another source for it; if we cannot find one, then the variable is part of an unfounded set.

We initialize the source pointers of founded variables before beginning search. Our initialization algorithm, shown in Figure 5.1, partitions the set of founded variables into three disjoint sets: $MustBeTrue$, the set of founded variables that are true in every stable model of the program; $MayBeTrue$, the founded variables that can be true in some stable model; and $CantBeTrue$, founded variables that cannot be true in any stable model. Variables in $MustBeTrue$ cannot be part of any unfounded set, and the unfounded variables in $CantBeTrue$ can be set to false at this early stage. Only variables in $MayBeTrue$ actually require source pointers; we record the source “pointers” of variables in $MustBeTrue$ as true, and the source pointers of variables in $CantBeTrue$ as false.

The pseudo-code in Figure 5.1 describes a bottom-up calculation which is similar to

\(^{11}\) $r^+$ here strictly represents the positive founded variables in the body, and does not include the standard variables.
the Dowling-Gallier algorithm (Dowling and Gallier, 1984). The algorithm keeps two queues of rules, \textit{MustBeQ} and \textit{MayBeQ}, that we use to incrementally build \textit{MustBeTrue} and \textit{MayBeTrue} respectively. If for some \( r \in \text{rules}(P) \), \( \theta(r) = \text{true} \) and \( r^+ \subseteq \text{MustBeTrue} \), then we add rule \( r \) to \textit{MustBeQ}. Otherwise, if \( \theta(r) \neq \text{false} \) and \( r^+ \subseteq \text{MustBeTrue} \cup \text{MayBeTrue} \), then we add \( r \) to \textit{MayBeQ}. Since the heads of a rule \( r \) in \textit{MayBeQ} can become true due to \( r \), we set their sources to \( r \). Whenever we set the source of a founded variable \( a \) equal to rule \( r \), we make the propagator subscribe to the event \( \text{ub.event(body}(r)) \) (\( \text{body}(r) \) becoming false), since if \( \text{body}(r) \) becomes false, the propagator must determine a new source for \( a \) (or construct an unfounded set from \( a \) if one exists).

The algorithm works by keeping a count \( ct(r) \) for each rule \( r \). Before the end of the first while loop, \( ct(r) \) represents the number of founded variables in \( r^+ \) that we need to find in \textit{MustBeTrue} before we can put the heads supported by \( r \) into \textit{MustBeTrue}. After the end of the first while loop, when there is no possibility left of finding any founded variables that must be true, \( ct(r) \) represents the number of founded variables in \( r^+ \) that we need to find in \textit{MayBeTrue} before we can put the heads supported by \( r \) into \textit{MayBeTrue}.

At the end, we set all the founded variables that do not have a source to false, since these are variables that cannot be true in any stable model of the program.
EstablishSourcePointers()

source(a) := false for each a ∈ V_F

for (each rule r)
    if (D(r) ≠ false) then ct(r) := |r^+| else ct(r) := ∞
    if (ct(r) = 0) then MustBeQ.add(r) else MaybeQ.add(r)

while (MustBeQ ≠ ∅)
    r := MustBeQ.pop()
    a := head(r)
    if (source(a) = false)
        source(a) := true
        for (c ∈ posInRules(a))
            ct(c) := ct(c) − 1
            if (ct(c) = 0)
                if (D(c) = true) then MustBeQ.add(c) else MaybeQ.add(c)

while (MaybeQ ≠ ∅)
    r := MaybeQ.pop()
    a := head(r)
    if (source(a) = false)
        source(a) := r, Subscribe(ub_event(body(r)))
        for (c ∈ posInRules(a))
            ct(c) := ct(c) − 1
            if (ct(c) = 0) then MaybeQ.add(c)

for (each variable a ∈ V_F for which source(a) = false)
    D(a) := false

Figure 5.1: Initializing source pointers

Example 5.2. Refer to the Constraint Answer Set Program in Example 5.1. At root, no propagation takes place, and we enter EstablishSourcePointers. The positive founded literals count for rules R_3, R_4, R_5 is equal to 0 while the count for rules R_1, R_2 is equal to 1. We put R_3, R_4, R_5 in
MayBeQ. Since MustBeQ is empty, we enter the second while loop.

We pop each rule from MayBeQ, and as a result, the source pointers are set as follows: source(c) = R₃, source(b) = R₄, and source(a) = R₅. When we pop R₄ and R₅, they also make the counts of R₁ and R₂ respectively equal to 0, and also put them in MayBeQ, but that has no effect since the sources of a and b have already been established.

5.4 Algorithm Of Anger et al. (2006)

In this section, we use the unfounded set described by Anger et al. (2006).

5.4.1 The WakeUp And Propagate Functions

EstablishSourcePointers() subscribes our propagator to events that record the source of a founded variable becoming false. When such events happen, the subscription system calls the WakeUp function in Figure 5.2, which delegates most of its work to the Propagate function in Figure 5.3. These two functions jointly manage two global variables: \( U \), which contains atoms that form an unfounded set, and \( Pend \), which contains atoms that are pending an unfounded check. These variables are global because they must retain their values across all the propagation invocations in a propagation step between two consecutive decision levels. We set both variables to be empty the first time we get control after any decision.

When an invocation of WakeUp tells us that body of \( r \) is false, if head of \( r \) is supported by it and the head is not false, we add it to the pending queue \( Pend \). However, we do not process the pending queue immediately; we let higher priority propagators run first, to allow them to tighten the current assignment as much as possible before we process the pending queue in our own low priority Propagate function.
WakeUp(ub_event(body(r)))
  if (first wakeup on current search tree branch)
    U := ∅, Pend := ∅
  if (source(head(r)) = r and D(head(r)) ≠ false)
    Pend.add(head(r))
    Queue()

Figure 5.2: Event notification for ANGER

Propagate()
    U := U \ {a ∈ V_F | D(a) = false}
    Pend := Pend \ {a ∈ V_F | D(a) = false}
  while (U = ∅)
    if (Pend = ∅) return
    a := Pend.pop()
    if (θ(a) ≠ false)
      if (∃r ∈ rules(a) : D(r) ≠ false and scc(a) ≠ scc(r))
        source(a) := r, Subscribe(ub_event(body(r)))
      else UnfoundedSet(a)
    let a ∈ U
    if (D(a) ≠ true) Requeue()
    Propagate D(a) = false with explanation loop nogood(U, a)

Figure 5.3: Propagation for ANGER

The Propagate function starts by removing all the founded variables that have become false from both U and Pend. (Other propagators with higher priorities can set a variable in Pend to false after the WakeUp that put that variable in Pend.)

If U is not empty, we remove an founded variable a from U, and set it to false with its loop nogood (2.4.5) as the explanation. In the case where it is already true, the solver
will detect this as failure, and the loop nogood will act as the conflict clause. If it is not already true, we requeue the propagator to allow propagators of higher priority efficiently propagate the effects of setting \( a \) to false. This may or may not fix the values of all the founded variables in the updated \( U \). While it does not, each invocation of Propagate will set another unfounded variable to false.

When there are no more known-to-be-unfounded founded variables left, we look for founded variables in the pending queue that can be part of an unfounded set. If the pending queue is empty, then there cannot be any more unfounded sets, and we are done. If there is a variable \( a \) in \( Pend \), we test whether it is supported by an external rule, a rule \( r \) in a different SCC. If it is, then \( a \) is not unfounded. If it isn’t, then it is possible that \( a \) is part of an unfounded set, and we invoke UnfoundedSet to check if \( a \) can be extended to an unfounded set. If the call fails and leaves \( U = \emptyset \), we try again with a different member of \( Pend \). If it succeeds, we handle the newly-made unfounded set the same way as we handle unfounded sets that already exist when Propagate is invoked.

### 5.4.2 Unfounded Set Calculation

The unfounded set algorithm for ANGER is given in Figure 5.4. The key local data structure in the unfounded set algorithm is \( Unexp \), which contains the rules that may contain external support for some of the founded variables in \( U \). A rule \( r \) can support its head \( a \) only if it is not false, and it does not contain any founded variables that are in \( U \). If \( r \) is in a different SCC than \( a \), then we take its assignment as a given; if it is not false, then it supports \( a \) and therefore \( a \) cannot be declared unfounded; if it is false, then it does not support \( a \). If it is in the same SCC as \( a \), then \( r \) may or may not support \( a \); we need to find out which. That is why we put into \( Unexp \) the set of rules that may support the founded variables in \( U \). To help us to do this, we define the function \( \text{maysupport}(a) \) as \( \{ r \mid r \in \text{rules}(a), \theta(r) \neq \text{false}, r^+ \cap U = \emptyset, \text{scc}(r) = \text{scc}(a) \} \). The algorithm uses two other data structures, \( SAtoms \) and \( SRules \) (\( S \) is for supported in this context): \( SAtoms \) contains founded variables that have been proven to be not unfounded, while \( SRules \) contains externally supported rules. A non-false rule \( r \) is externally supported if every founded variable in \( r^+ \) is either in \( SAtoms \), or belongs to a different SCC.
The algorithm processes the unexplored rules in Unexp one by one. It looks at the positive founded variables in each such rule. Those that are in SAtoms or in a different SCC are known to support r; the others are not. We compute \( nks(r, curscc, SAtoms) \) as the set of not-known-to-be-supporting atoms in \( r \): \( nks(r, curscc, SAtoms) = \{ p \mid p \in r^+ : p \notin SAtoms \text{ and } scc(p) = curscc \} \).

If this is not empty, then we need to test the founded variables in it to see whether or not they actually do support \( r \). If there is a non-false rule with \( p \) as head that is either external or in SRules, then we have found a source for \( p \). The algorithm records this source if necessary. It then removes \( p \) from Pend and adds it to the set of supported founded variables. If we cannot find a potential source (external or already in SRules for \( p \)), then the algorithm makes \( p \) part of the unfounded set \( U \). The definition of Unexp says that rules whose positive founded variables are in \( U \) must not be in it; therefore, we remove from it the bodies that would now violate that invariant. To allow later iterations of the outermost loop to check whether \( p \) can be supported via other bodies, we then add those possible bodies to Unexp. All these changes may have reduced the set of not-known-to-be-supporting atoms to the empty set, which is why we compute that set again.

If the set of not-known-to-be-supporting atoms is empty, either originally or after being recomputed, then we know \( r \) is externally supported. This means that if head of \( r \) is in \( U \), then it is now supported by \( r \).

We compute \( A \) as the set of such newly justified variables, and initialize \( A \) with head(\( r \)). We record source of head(\( r \)) as \( r \), remove the head from \( U \) and Pend, and add it to SAtoms. Adding it to SAtoms may make more rules qualify for membership of SRules, which in turn may provide external support for more founded variables. We put any such atoms into \( A \) as well, and we keep going until everything in \( A \) has been processed. Once we have removed as many atoms as possible from \( U \) and have reached a fixpoint, we reinitialize Unexp based on the final value of \( U \).
UnfoundedSet(a)
  \texttt{curscc} := \texttt{scc}(a)
  \texttt{U} := \{a\}
  \texttt{Unexp} := \texttt{maysupport}(a)
  \texttt{SAtoms} := \emptyset, \texttt{SRules} := \emptyset

\textbf{while} (\texttt{Unexp} \neq \emptyset)
  \texttt{r} := \texttt{Unexp.pop()}
  \textbf{if} (\texttt{nks}(r, \texttt{curscc}, \texttt{SAtoms}) \neq \emptyset) \quad \% \texttt{r} \text{ is not externally supported}
    \textbf{for} (\texttt{p} \in \texttt{nks}(r, \texttt{curscc}, \texttt{SAtoms}))
      \textbf{if} (\exists \texttt{c} \in \texttt{rules}(\texttt{p}) : \texttt{D}(\texttt{c}) \neq \texttt{false} \text{ and } (\texttt{scc}(\texttt{c}) \neq \texttt{curscc} \text{ or } \texttt{c} \in \texttt{SRules}))
        \textbf{if} (\texttt{scc}(\texttt{source}(\texttt{p})) = \texttt{curscc})
          \texttt{source}(\texttt{p}) := \texttt{c}, \texttt{Subscribe}((\texttt{ub.event(\texttt{body}(\texttt{c}))}))
          \texttt{Pend.remove}(\texttt{p})
          \texttt{SAtoms.add}(\texttt{p})
      \textbf{else}
        \texttt{U.add}(\texttt{p})
        \texttt{Unexp} := \texttt{Unexp} \setminus \{d \mid d \in \texttt{Unexp}, p \in d^+\}
        \texttt{Unexp} := \texttt{Unexp} \cup \texttt{maysupport}(p)
  \textbf{else}
    \texttt{U.add}(\texttt{p})
  \texttt{Unexp} := \texttt{Unexp} \setminus \{d \mid d \in \texttt{Unexp}, p \in d^+\}
  \texttt{Unexp} := \texttt{Unexp} \cup \texttt{maysupport}(p)

\textbf{if} (\texttt{nks}(r, \texttt{curscc}, \texttt{SAtoms}) = \emptyset) \quad \% \texttt{r} \text{ is externally supported}
  \texttt{SRules.add}(\texttt{r})
  \texttt{A} := \{\texttt{head}(\texttt{r})\}
  \texttt{source(\texttt{head}(\texttt{r}))} = \texttt{r}, \texttt{Subscribe}((\texttt{ub.event(\texttt{body}(\texttt{r}))})

\textbf{while} (\texttt{A} \neq \emptyset)
  \texttt{a} := \texttt{A.pop()}
  \texttt{U.remove}(\texttt{a})
  \texttt{Pend.remove}(\texttt{a})
  \texttt{SAtoms.add}(\texttt{a})
  \textbf{for} (\texttt{j} \in \texttt{posInRules}(\texttt{a}) : \texttt{D}(\texttt{j}) \neq \texttt{false} \text{ and } \forall t \in j^+, (t \in \texttt{SAtoms} \text{ or } \texttt{scc}(t) \neq \texttt{curscc}))
    \texttt{SRules.add}(\texttt{j})
    \textbf{if} (\texttt{head}(\texttt{j}) \in \texttt{U})
      \texttt{source(\texttt{head}(\texttt{j}))} := \texttt{j}, \texttt{Subscribe}((\texttt{ub.event(\texttt{body}(\texttt{j}))})
      \texttt{A.add(\texttt{head}(\texttt{j}))}
  \texttt{Unexp} := \bigcup_{p \in \texttt{U}} \texttt{maysupport}(p)

Figure 5.4: Unfounded set algorithm for \texttt{ANGER}: (source Anger et al. (2006))
Example 5.3. For the CASP in Example 5.1, we described how we established source pointers at root in Example 5.2. In this example, we show how the functions WakeUp, Propagate, and UnfoundedSet work for two decisions for ANGER. We begin with the sources: source(c) = R₃, source(b) = R₄, and source(a) = R₅.

Decision: \( D(x₁) = 5 \)

Say the solver decides \( D(x₁) = 5 \). Due to constraint C₂, we propagate \( D(s) = false \), which propagates \( body(R₄) = false \). Since \( R₄ \) is the source of \( b \), this triggers \( WakeUp(ub\_event(body(R₄))) \). Since \( b \) is not false, we put it in Pend.

When we enter Propagate, we pop \( b \) from Pend. Since there is no non-false external rule for \( b \), we call UnfoundedSet(\( b \)). In the unfounded set function, we initialize \( U \) with \{\( b \}\}, and \( Unexp \) with \{\( R₂ \}\}, since that is the only rule with \( b \) as head that can potentially support it. Then, we enter the first while loop. We remove \( R₂ \) from \( Unexp \), and the not-known-to-be-supported set of founded variables for \( R₂ \) includes \( a \). But next, we immediately discover that \( a \) is supported by an external rule and therefore put it in SAtoms. As a result, the not-known-to-be-supporting set for \( R₂ \) is now empty, and we enter the if condition in the lower half of the algorithm. We add \( R₂ \) in supported rules, and make it the source for \( b \), i.e. source(\( b \)) = \( R₂ \). When we enter the while loop that follows, we remove \( b \) from Pend as well as from \( U \). In the following for loop, \( b \) makes \( R₁ \) a supported rule, but that has no effect since \( a \) already has a support, and is not in \( U \). Finally, we update \( Unexp \) which is now empty, and therefore, we exit the function.

When we resume Propagate, since \( U \) is empty, and Pend is empty, we exit the function.

Decision: \( D(x₂) = 5 \)

Say the solver now decides \( D(x₂) = 5 \). Due to constraint C₁, since \( x₁ \) and \( x₂ \) have the same value, we propagate \( D(t) = false \), which propagates \( body(R₅) = false \). Since \( R₅ \) is the source of \( a \), this triggers \( WakeUp(ub\_event(body(R₅))) \). Since \( a \) is not false, we put it in Pend.

When we enter Propagate, we pop \( a \) from Pend. Since there is no non-false external rule for \( a \), we call UnfoundedSet(\( a \)). In the unfounded set function, we initialize \( U \) with \{\( a \}\}, and
Unexp with \{R_1\}, since that is the only rule with \(a\) as head that can potentially support it. Then, we enter the first while loop. We remove \(R_1\) from Unexp, and the not-known-to-be-supported set of founded variables for \(R_1\) includes \(b\). Since there is no non-false rule for \(b\) which is either external or in SRules, we add \(b\) to UI. There is no rule in maysupport\((b)\) since the only candidate \(R_2\) includes \(a\) in its positive literals. Therefore, \(U = \{a, b\}\) is an unfounded set and we exit the function.

When we resume Propagate, let us say we read \(b\) in \(U\). Since \(D(b) \neq \text{true}\), we requeue the propagator since we are about to set it to false, and would like to run the higher priority propagators first (more particularly, unit propagation on completion clauses), before we continue setting other founded variables in \(U\) to false, if they already haven’t been set false by higher priority propagators. Then we propagate \(D(b) = \text{false}\) with the explanation: \((-b \lor \text{body}(R_4) \lor \text{body}(R_5))\).

Setting \(b\) to false propagates \(D(\text{body}(R_1)) = \text{false}\), and due to completion, \(D(a) = \text{false}\), which propagates \(D(\text{body}(R_3)) = \text{true}\), which propagates \(D(c) = \text{true}\). \(D\) is now an assignment, and a stable one, according to Example 5.1.

### 5.5 Algorithm Of Gebser et al. (2012b)

In this section, we describe the unfounded set algorithm of Gebser et al. (2012b).

#### 5.5.1 The WakeUp And Propagate Functions

Like the previous algorithm (ANGER), this algorithm also uses the same two global variables, \(\text{Pend}\) and \(U\). There is no change in the WakeUp function, it is the same as the previous WakeUp given in Figure 5.2. The pseudo-code for the Propagate function is given in Figure 5.5. After filtering out the founded variables that have already been set false from \(U\) and \(\text{Pend}\), it checks whether \(U\) is empty (it could be non-empty due to previous values found by the unfounded set algorithm that yet have not been set to false). If so, and if \(\text{Pend}\) is also empty, there are no more non-false unjustified variables. If \(\text{Pend}\) is not empty, we call UnfoundedSet function, which unlike the one in Figure 5.4, does not take a founded variable in its argument. The function is given in the next subsection.

Next, we check whether \(U\) is non-empty, either as an else branch for the previous
Propagate()

\[
U := U \setminus \{ a \in \mathcal{V}_F \mid D(a) = false \}
\]

\[
Pend := Pend \setminus \{ a \in \mathcal{V}_F \mid D(a) = false \}
\]

\[
\text{if } (U = \emptyset) \]

\[
\text{if } (Pend = \emptyset) \text{ then return else UnfoundedSet()}
\]

\[
\text{if } (U \neq \emptyset)
\]

\[
\text{let } a \in U
\]

\[
\text{if } (D(a) \neq true) \text{ Requeue()}
\]

\[
\text{Propagate } D(a) = false \text{ with explanation loop nogood}(U, a)
\]

Figure 5.5: Propagation for GEBSER

5.5.2 Unfounded Set Calculation

The unfounded set algorithm for GEBSER is given in Figure 5.6. The algorithm can be divided into two parts. In the first part, the scope S of the unfounded sets is defined; by definition, this is potentially the maximal unfounded set, constructed as follows. It is initialized with variables in Pend, as these are the variables that have lost their justifications since the last unfounded set check, but that have not yet been falsified by propagation. The scope is then expanded using a fixed point calculation by including all other variables that can be reached in the justification graph from any variable in the scope. More concretely, in each round, the set T contains each non-false variable, not yet in S, whose source has at least one positive founded variable in S. The idea is that since all variables in the scope have lost their justification, the bodies in which they appear positively cannot be valid justifications.

In the second loop, the unfounded set U is constructed (or justifications readjusted if none exists). We run this loop until we have exhausted all atoms in the scope S (and
consequently, also in $Pend$). $U$ is initialized with some variable $p$ in the scope. Then the following is repeated until either $U$ is empty, or all the external rules of $U$ have false bodies in which case $U$ is an unfounded set that is returned (and this entire computation runs on $scc(p)$). A non-false rule $r$ that is external to $U$ is picked. Then $A$ contains all positive founded variables in $r$ that are in the same SCC and also in the scope. If $A$ is empty, then rule $r$ is completely justified, so its head is removed from $U$, $S$, and $Pend$, and the source of $head(r)$ is adjusted. If $A$ is non-empty, then it is appended to $U$.

**UnfoundedSet()**

\[
S := Pend
\]

repeat

\[
T := \{ p \in \mathcal{V} : D(p) \neq false, p \notin S, (\exists p' \in S : scc(p') = scc(p), source(p) \in posInRules(p')) \}
\]

$S := S \cup T$

until $(T = \emptyset)$

while $(S \neq \emptyset)$, let $p \in S$

$U := \{ p \}$

while $(U \neq \emptyset)$

if $(D(r) = false$ for all $r \in ext(U))$ return $U$

let $r \in ext(U)$ s.t. $D(r) \neq false$

$A = \{ a \in r^+ : scc(a) = scc(p), a \in S \}$

if $(A = \emptyset)$

source($head(r)$) := $r$, Subscribe($ub\_event(body(r))$)

$U.remove(head(r))$

$Pend.remove(head(r))$

$S.remove(head(r))$

else $U := U \cup A$

Figure 5.6: Unfounded set algorithm for GEBSER (source: Gebser et al. (2012b))

**Example 5.4.** For the CASP in Example 5.1, we described how we established source pointers
at root in Example 5.2. In this example, we show how the functions \texttt{WakeUp}, \texttt{Propagate}, and \texttt{UnfoundedSet} work for two decisions for GEBSER. We begin with the sources: \texttt{source(c) = R}_3, \texttt{source(b) = R}_4, and \texttt{source(a) = R}_5.

\textbf{Decision: } D(x_1) = 5

\textit{Say the solver decides } D(x_1) = 5. \textit{Due to constraint } C_2, \textit{we propagate } D(s) = \texttt{false}, \textit{which propagates } body(R_4) = \texttt{false}. \textit{Since } R_4 \textit{is the source of } b, \textit{this triggers } \texttt{WakeUp} ub\_event(body(R_4))). \textit{Since } b \textit{is not false, we put it in Pend.}

\textit{When we enter } Propagate, \textit{we call } UnfoundedSet. \textit{In the unfounded set function, the scope } S \textit{is initialized with Pend, i.e., } S = \{b\}. \textit{Since } b \textit{does not appear positively in any other current valid source, no scope extension takes place, and we enter the while loop. We initialize } U \textit{with } \{b\}, \textit{since } U \textit{has an external rule that is not false, namely } R_2, \textit{we inspect it. There is no intersection of positive founded variables in } R_2 \textit{and the scope } S, \textit{therefore, we change the source of } b \textit{to } R_2, \textit{and remove } b \textit{from Pend, } U, \textit{and } S. \textit{Since } S \textit{is now empty, we exit the function.}

\textit{When we resume } Propagate, \textit{since } U \textit{is empty, we exit the function.}

\textbf{Decision: } D(x_2) = 5

\textit{Say the solver now decides } D(x_2) = 5. \textit{Due to constraint } C_1, \textit{since } x_1 \textit{and } x_2 \textit{have the same value, we propagate } D(t) = \texttt{false}, \textit{which propagates } body(R_5) = \texttt{false}. \textit{Since } R_5 \textit{is the source of } a, \textit{this triggers } \texttt{WakeUp} ub\_event(body(R_5))). \textit{Since } a \textit{is not false, we put it in Pend.}

\textit{When we enter } Propagate, \textit{we call } UnfoundedSet. \textit{In the unfounded set function, the scope } S \textit{is initialized with Pend, i.e., } S = \{a\}. \textit{Since } a \textit{does appear positively in the source of } b, \textit{which is } R_2, \textit{the scope is extended to include } b. \textit{No other extension happens, and we leave the first repeat-until loop with } S = \{a,b\}. \textit{Next, we enter the while loop, say we initialize } U = \{a\}. \textit{Since the external rules for } a \textit{include } R_1, \textit{we inspect it. In the set } A \textit{in the pseudo-code, we get } \{b\} \textit{since it is in the same SCC as } a \textit{and it is also in the scope. We add } b \textit{to } U, \textit{and now, } U = \{a,b\}. \textit{We find that all external rule bodies for } U \textit{are false, therefore we return since we have found an unfounded set.}
5.6 Experiments

We benchmarked our implementations ANGER and GEBSER against two competing systems. The first is a combination of CLASP (version 2.0.6) and GRINGO (version 3.0.4), which we call CL+GR in our tables for brevity. The second is CLINGCON (Gebser et al., 2009c) (version 2.0.0-beta), which is an extension of CLASP with CP capabilities. We ran all the benchmarks on a Lenovo model 3000 G530 notebook with a 2.1 GHz Core 2 Duo T6500 CPU and 3 GB of memory running Ubuntu 12.04. We repeated each experi-

<table>
<thead>
<tr>
<th></th>
<th>Solved</th>
<th>Opt</th>
<th>AvgPct</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RoutingMin</strong></td>
<td>CL+GR</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>CLINGCON</td>
<td>19</td>
<td>3</td>
<td>69.1</td>
</tr>
<tr>
<td>GEBSER</td>
<td>19</td>
<td>8</td>
<td>33.5</td>
</tr>
<tr>
<td>ANGER</td>
<td>18</td>
<td>9</td>
<td>33.9</td>
</tr>
<tr>
<td><strong>RoutingMax</strong></td>
<td>CL+GR</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>CLINGCON</td>
<td>22</td>
<td>0</td>
<td>100.0</td>
</tr>
<tr>
<td>GEBSER</td>
<td>27</td>
<td>0</td>
<td>42.2</td>
</tr>
<tr>
<td>ANGER</td>
<td>27</td>
<td>0</td>
<td>37.4</td>
</tr>
</tbody>
</table>

Table 5.1: Results for routing on 34 instances

When we resume Propagate, since \( U \neq \emptyset \), let us say we read \( b \) in \( U \). Since \( D(b) \neq \text{true} \), we enter the second if condition in the function. We requeue the propagator, and then we propagate \( D(b) = \text{false} \) with the explanation: \(( \neg b \lor \text{body}(R_4) \lor \text{body}(R_5))\).

Setting \( b \) to false propagates \( D(\text{body}(R_1)) = \text{false} \), and due to completion, \( D(a) = \text{false} \), which propagates \( D(\text{body}(R_3)) = \text{true} \), which propagates \( D(c) = \text{true} \). \( D \) is now an assignment, and a stable one, according to Example 5.1.

5.6 Experiments

When we resume Propagate, since \( U \neq \emptyset \), let us say we read \( b \) in \( U \). Since \( D(b) \neq \text{true} \), we enter the second if condition in the function. We requeue the propagator, and then we propagate \( D(b) = \text{false} \) with the explanation: \(( \neg b \lor \text{body}(R_4) \lor \text{body}(R_5))\).

Setting \( b \) to false propagates \( D(\text{body}(R_1)) = \text{false} \), and due to completion, \( D(a) = \text{false} \), which propagates \( D(\text{body}(R_3)) = \text{true} \), which propagates \( D(c) = \text{true} \). \( D \) is now an assignment, and a stable one, according to Example 5.1.
ment with a timeout five times, and each experiment without a timeout twice; the results we present are their averages.

We ran two sets of benchmarks. The first set consists of different instances of two routing problems, which are based on the models used in the experiments by Liu et al. (2012). Our reason for selecting these two problems is that they involve not just reachability, but also variables with large finite domains. The two problems differ only in their objective; they use the same data representation and impose the same set of constraints. Each instance of these problems is specified by

- a weighted directed graph \((V, E, w)\) where \(w : E \mapsto \mathbb{N}\),
- a source node \(s \in V\),
- a set of destination nodes \(D \subseteq V \setminus \{s\}\), and
- a deadline for each destination \(f : D \mapsto \mathbb{N}\).

Their solutions consist of two parts:

- a cycle-free route \((r_0, r_1, \ldots, r_k)\) where \(r_0 = s, (r_i, r_{i+1}) \in E\) for all \(i \in \{1, \ldots, k - 1\}\),
  and for each \(d \in D\), \(d = r_i\) for some \(i \in \{1, \ldots, k\}\), and
- a time assignment \(t : V \mapsto \mathbb{N}\) such that \(t(r_0) = 0, t(r_{i+1}) \geq t(r_i) + w(r_i, r_{i+1})\) for all \(i \in \{1, \ldots, k - 1\}\), and for each \(d \in D\), \(t(d) \leq f(d)\).

For the \textit{RoutingMin} problem, the objective is minimizing the total delay \(\sum_{d \in D} (f(d) - t(d))\). For the \textit{RoutingMax} problem, the objective is maximizing the total delay.

Table 5.1 presents our results on 34 instances each of RoutingMin and RoutingMax. The graphs in the instances are random and their sizes range from 21 to 87 nodes. The Solved column gives the number of instances for which the named solver computed a result (which may or may not be optimal) within the timeout period, which was one minute. The Opt column gives the number of these instances for which the solver not only computed the optimal result, but also proved it to be optimal.

Since the sizes of the instances vary significantly, the minimum and maximum values of the total delay differ greatly as well. Averages of the delays are therefore not an appropriate representation of the overall quality of the solutions from a solver. Therefore we
### Table 5.2: ASP problems, geometric restart, no timeout

<table>
<thead>
<tr>
<th>Problem</th>
<th>CL+GR</th>
<th>flat</th>
<th>ANGER</th>
<th>GEBSER</th>
</tr>
</thead>
<tbody>
<tr>
<td>WR (S/8)</td>
<td>894.49</td>
<td>10.22</td>
<td>166.74</td>
<td>95.39</td>
</tr>
<tr>
<td>WR (U/7)</td>
<td>24.14</td>
<td>8.89</td>
<td>41.20</td>
<td>28.44</td>
</tr>
<tr>
<td>GP (S/7)</td>
<td>9.69</td>
<td>2.88</td>
<td>656.69</td>
<td>4.13</td>
</tr>
<tr>
<td>GP (U/6)</td>
<td>43.02</td>
<td>0.95</td>
<td>221.38</td>
<td>9.76</td>
</tr>
<tr>
<td>CDS (S/7)</td>
<td>26.80</td>
<td>0.39</td>
<td>74.98</td>
<td>44.53</td>
</tr>
<tr>
<td>CDS (U/8)</td>
<td>1618.22</td>
<td>0.64</td>
<td>566.47</td>
<td>318.65</td>
</tr>
<tr>
<td>MG (S/15)</td>
<td>2.56</td>
<td>32.95</td>
<td>43.04</td>
<td>42.14</td>
</tr>
</tbody>
</table>

Express the quality of each solution as a percentage of the maximum delay computed by any solver on the relevant problem instance. If all the solvers compute a delay of 0, we score all solvers as 0% for RoutingMin and as 100% for RoutingMax. The AvgPct column shows the average of these percentages for the instances for which all the solvers get a solution. All the solvers were run with a slow restart strategy that used a Luby sequence (Luby et al., 1993) with a restart base of 400.

Due to the large domains involved, grounding is very inefficient, which causes CL+GR to run out of memory on all our test instances, even the smallest. For RoutingMin, all the solvers solve roughly the same number of instances, but ANGER and GEBSER get optimal solutions on almost three times as many instances, and the average quality of their solutions is also better by about a factor of 2. (For minimization, better performance is represented by smaller percentages, while for maximization, it is represented by larger ones.) For the instances of RoutingMax that all the solvers can solve, CLINGCON invariably generates the best solutions. However, CLINGCON generates solutions for substantially fewer instances than ANGER and GEBSER, showing that it is not as robust.

Our second set of benchmarks is a selection of problems taken from the second ASP competition[^12]: Wire Routing (WR), Graph Partitioning (GP), Connected Dominating Set

Table 5.3: ASP problems, Luby restart, with timeout

<table>
<thead>
<tr>
<th>Problem</th>
<th>CL+GR</th>
<th>flat</th>
<th>ANGER</th>
<th>GEBSER</th>
</tr>
</thead>
<tbody>
<tr>
<td>WR (S/8)</td>
<td>148.21 (5/1) 9.86</td>
<td>149.12 (4/1) 96.74 (5/1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WR (U/7)</td>
<td>101.68 (5/1) 11.68</td>
<td>36.68 (0/0) 49.10 (0/0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GP (S/7)</td>
<td>39.24 (0/0) 2.58</td>
<td>21.36 (0/0) 46.52 (0/0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GP (U/6)</td>
<td>169.22 (5/1) 0.91</td>
<td>123.96 (3/1) 121.72 (0/0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDS (S/7)</td>
<td>185.09 (10/2) 0.41</td>
<td>102.79 (5/1) 127.88 (5/1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDS (U/8)</td>
<td>274.77 (10/2) 0.69</td>
<td>321.20 (20/4) 314.56 (20/4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MG (S/15)</td>
<td>5.07 (0/0) 52.86</td>
<td>49.95 (5/1) 1.37 (0/0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(CDS), and Maze Generation (MG). For each of these problems, we report on their satisfiable (S) and unsatisfiable (U) instances in separate rows of both Tables 5.2 and 5.3; the numbers after the forward slashes give the number of instances in each category. Table 5.2 shows results for a setup in which all the solvers were run using the default restart strategy of CLASP (geometric restart, with the restart threshold starting at 100 conflicts, multiplied by 1.5 after each restart) and without time limits, while Table 5.3 shows the results when all the solvers were run with a Luby sequence restart strategy with restart base 10, and with a timeout of 10 minutes.

The numbers in the slots of Table 5.2 all represent an average runtime, in seconds, over all the problem instances represented by the row. The CL+GR column gives the average time taken by CL+GR to solve those instances. The flat column gives the average time taken to flatten the MiniZinc model to FlatZinc. The ANGER and GEBSER overall (o/a) columns give the average time taken to solve the resulting FlatZinc models using the ANGER and GEBSER variants of our implementation. The ANGER and GEBSER prop columns give the average times taken by our propagator within those overall solution times.

The numbers in time slots of Table 5.3 also represent average execution times; the execution time will be the timeout time (600 seconds) if the solver does not complete before then. Table 5.3 omits propagation times to make room for the numbers in parentheses.
5.6 Experiments

after each average execution time. These represent respectively the number of runs on
which the given solver failed to produce a solution before the timeout, and the number
of instances to which those runs belong. (For example, 4/1 means that of the five runs
on a problem instance, one produced a solution, but four did not.) Neither table has a
column for CLINGCON, since the purpose of Tables 5.2 and 5.3 is to compare GEBSER and
ANGER with a native ASP solver on pure Boolean problems. The results of running these
problems on CLINGCON would be the same as the results of CL+GR, since these problems
have nothing to do with the difference between CL+GR and CLINGCON, namely the finite
domain extension present in CLINGCON.

The overall results in Table 5.2 are mixed. On WR/U, GP/S, GP/U and (due to flat-
tening) MG/S, CL+GR clearly beats both ANGER and GEBSER. On WR/S and CDS/U,
both ANGER and GEBSER clearly beat CL+GR. On CDS/S, CL+GR clearly beats ANGER,
but edges out GEBSER by just a whisker. The overall winner on these tests is clearly
CL+GR.

However, this picture changes when we switch our attention to Table 5.3. Even after
including flattening time, GEBSER is faster than CL+GR on four of the seven problem
sets (WR/S, WR/U, GP/U, CDS/S), and it is slower on only three (GP/S, CDS/U and
MG/S). It is also more robust, failing to find a solution on only six problem instances,
compared to seven for CL+GR. Our other system ANGER is less robust, failing to find
solutions on eight problem instances, though for two of these, it did solve them on some
runs. However, to compensate for this, ANGER is the fastest system on three problem sets

The propagation time columns in Table 5.2 show that ANGER spends a lot more time
on unfounded set propagation than GEBSER. In some cases, such as WR/S, this pays off
handsomely, in the form of more effective pruning of the search space. In some other
cases, such as CDS/U, ANGER spends less time outside the propagator than GEBSER, so
ANGER seems to get better pruning, but not enough to pay back the extra cost of the
propagator itself. And on most problems in Table 5.2, the extra cost of its propagator
does not even help ANGER get better pruning. This suggests that we should investigate
whether one can blend the two unfounded set calculation algorithms in order to achieve
the pruning power of ANGER (Anger et al., 2006), or something close to it, at an efficiency closer to that of GEBSER (Gebser et al., 2012b).

5.7 Related Work

There are several approaches in the literature that aim at removing the standard grounding bottleneck from ASP systems. A majority of these approaches follow the same strategy of introducing auxiliary standard variables (constraint atoms) to augment a logic program with constraints. More particularly, they introduce finite domain integer variables and constraints inside the ASP program. The ASP solver passes these to a CP solver while maintaining a Boolean variable to represent the truth value of each constraint that is in the ASP program. For a constraint $c$ that appears in the program, this is done by reifying the constraint, i.e., introducing a Boolean variable $b$ to represent whether the constraint is true. The ASP and CP solvers communicate using these introduced Boolean variables. E.g., if $b$ is set true by the ASP solver, then the constraint $c$ is enforced by the CP solver. Since the ASP solver treats CP as a blackbox, it cannot directly learn clauses from the propagation performed by the CP solver. Examples of systems that use this approach include: the AC SOLVER algorithm (Mellarkod et al., 2008), CLINGCON (Gebser et al., 2009c) and EZCSP (Balduccini, 2009).

Recently, some systems have been introduced that overcome the limited learning by using a single solver that supports both founded Booleans as well as standard integer variables and constraints over them. One way to achieve this is to introduce standard integer variables inside an ASP solver, and extending ASP’s propagation engine to work like a CP solver (Drescher and Walsh, 2012). There are two merits of our approach as compared to this approach. Firstly, we only require the implementation of a single propagator in a CP solver to support founded Booleans and normal rules. This is naturally supported by the general architecture of CP solvers which allows for implementation of multiple propagators. Second, the Global Constraint Catalog (GCC) \(^{13}\) lists more than 350 global constraint and for most of these constraints, the majority of constraint solvers

\(^{13}\)http://www.emn.fr/z-info/sdemasse/gccat/
either implement a dedicated propagator or rely on the decomposition provided in constraint language libraries like that of MINIZINC.\footnote{http://www.minizinc.org/downloads/doc-1.6/mzn-globals.html} On the other hand, ASP solvers, like SAT solvers, are designed for a fixed number of propagators. Moreover, there are no off-the-shelf decompositions of global constraints in ASP languages like GRINGO.

Translating in terms of its supported features a specification that is missing in a system is another way to remove standard grounding bottleneck. There is work on translating from an ASP program augmented with numeric variables and linear constraints to a Mixed Integer Program (Liu et al., 2012). As discussed earlier, the non-recursive parts of the program are straight-forward to translate. The non-trivial part is encoding rules that involve positive recursion. This is done using the level ranking mapping as given by Janhunen (2004). The fundamental idea of the translation is that if there is an unfounded set in the solution of the original program, then the mapping contains an inconsistent set of inequalities.

\section*{5.8 Conclusion}

We have shown a method for adding answer set programming capabilities to a lazy clause generation solver. We have shown two implementations of our extensions to CHUFFED. Our benchmark results show that both these systems can solve combined ASP/CP problems that cannot be solved by CL+GR, even though its ASP component, CLASP, won several competitions for pure ASP solvers. We have also shown that our system is competitive with CLINGCON, an extension of CLASP, on such problems, being better on some tasks, worse on others.
Chapter 6
Bound Founded Answer Set Programming

In Chapter 4, we described the motivation behind Bound Founded Answer Set Programming. We saw how founded grounding bottleneck makes ASP and CP poor choices for modeling e.g. shortest paths in a graph which requires foundedness over upper bounds of numeric variables. In BFASP, we can represent and reason directly with founded numeric variables. In this chapter, we formally define BFASP and its semantics. We also describe how ASP and many of its extensions can be translated to BFASP, thus establishing BFASP as a generalization of all such formalisms.

A logic program is composed of normal rules, and a normal rule is completely defined by its three constituents: head, positive set of literals and negative set of literals. The rules of BFASP generalize normal rules, and the core idea is generalizing the positive and negative variables. To motivate this, let us view a normal rule as an implication in the propositional sense, and point out an obvious property about the relation between the truth value of a body literal and the truth value of the implication. Changing the truth value of a positive variable in the body from false to true can only ever change the truth value of the implication to false. Similarly, changing the value of a negative variable from true to false has the same relation. In other words, the (truth value of) the implication is monotonically decreasing with respect to (truth value of) any positive literal in the body, and the implication is monotonically increasing w.r.t. any negative literal. The implication is also monotonically increasing w.r.t. the head. The central generalization of BFASP is that the implication is merely one form of a rule. There can be many other rule forms that are structurally similar to normal rules when we view their structure w.r.t. monotonocities of variables in them. Furthermore, these rules do not need to be restricted to Boolean
variables, but they can be extended to include numeric variables as well.  

Let us define monotonicity formally now.

### 6.1 Monotonicity And Positive Constraints

**Definition 6.1** (Monotonicity). A function \( f(x_1, \ldots, x_i, \ldots, x_n) \) is monotonically increasing (resp. decreasing) w.r.t. \( x_i \), written \( f_{\uparrow}^i \) (resp. \( f_{\downarrow}^i \)), iff \( f(k_1, \ldots, k_i, \ldots, k_n) \geq f(k_1, \ldots, k_i', \ldots, k_n) \) whenever \( k_i \geq k_i' \) (resp. \( k_i \leq k_i' \)).

Given a constraint \( c \) and a variable argument \( x \), \( c \) is monotonically increasing (resp. decreasing) w.r.t. \( x \), written \( c_{\uparrow}^y \) (resp. \( c_{\downarrow}^y \)), iff increasing (resp. decreasing) \( x \)'s value can never cause \( c \) to go from satisfied to unsatisfied.

We assume that the initial domains are taken into account while discussing monotonicity, e.g., \( |x| \geq 5 \) is non-monotonic in \( x \) in general, but if we know that \( x \) is non-negative, then the constraint is increasing in \( x \).

**Definition 6.2** (Positive constraints). A constraint \( c \) is positive iff it is monotonically increasing in exactly one of its arguments and is monotonically decreasing in all other arguments.

A collection of positive constraints is a positive CP.

**Definition 6.3** (Minimum assignment). The minimum assignment to a positive-CP \( P \), written \( \text{MA}(P) \), is a solution \( \theta \) such that there does not exist another solution \( \theta' \) s.t. \( \exists v \in V, \theta'(v) < \theta(v) \).

Similarly to how the least model for a positive logic program can be found by running unit propagation on the rules to fixed point and then setting all unfixed variables to false, the minimum solution for a positive CP can be found by running bounds consistent propagators on the constraints to fixed point and then setting all variables to their lowest allowed values.

---

15In this thesis, we restrict BFASP variables to Booleans, \( \mathbb{Z} \) and \( \mathbb{R} \), although there is no reason why the semantics and algorithms of BFASP cannot easily be extended to any total-ordered domain. We leave the extensions of semantics and algorithms to partially ordered domains for future work.
Example 6.1. Consider the constraints $x \geq 0$, $y \geq 4 + x$, $2x \geq y$. This is a positive CP, each constraint monotonically increasing in the first argument. A fixpoint calculation yields $x \geq 0$, $y \geq 4$, $x \geq 2$, $y \geq 6$, $x \geq 3$, $y \geq 7$, $x \geq 4$, $y \geq 8$. The minimum solution is $x = 4$, $y = 8$. □

The next proposition establishes that minimum assignments for positive CPs generalize least models of logic programs.

Proposition 6.1. Least models of positive logic programs is a special case of minimum models of positive set of constraints.

Proof. First, a positive logic program is a positive set of constraints, since each positive rule is logically a positive constraint by our definition. The definitions of least model and minimum model then coincide once we consider the ordering $true > false$ over Boolean variables. □

Note that unlike positive logic programs, it is possible for a positive CP to not have any minimum assignment at all. E.g. if $y$ is a founded real variable, then $y > 2$ has no minimum assignment.

6.2 Bound Founded Answer Set Programs

We now define bound founded answer set programs (BFASPs). A BFASP $P$ is a tuple $P = (V, C, R)$, where $V_S$ are standard variables, $V_F$ are founded variables, $C$ is a set of constraints, and $R$ is a set of rules. The set of variables can either be Boolean, integer, or real. Founded variables can be divided into two categories: lb-founded and ub-founded. Therefore, the complete type of a variable is determined by two factors: whether it is Boolean, integer, or continuous and whether it is lb-founded, ub-founded or standard. We can get the set of rules (constraints) for $P$ by $rules(P) (cons(P))$.

6.2.1 BFASP Rules

Each rule constraint $r$ is a pair $(c, y)$ where $c$ is a constraint over standard and founded variables in the problem, $y \in V_F$ is called the head variable of $r$ and $c$ is increasing in $y$. 
The constraint and head of a rule $r$ can be accessed by the functions $\text{cons}(r)$ and $\text{head}(r)$ respectively.

The stable model semantics requires that the lower bounds (resp. upper bounds) of lb-founded (resp. ub-founded) variables are justified by some rule constraint in order to be valid in a stable model. This means that an lb-founded Boolean variable is the same as a variable in ASP, i.e., in the absence of any rule that forces it to be true, the variable becomes false. We can intuitively think of an lb-founded integer $x$ with domain $[L \ldots U]$ as a set of ASP variables $[x \geq L], [x \geq L + 1], \ldots, [x \geq U], [x \geq U + 1]$ where $[x \geq v]$ represents the truth value of $x \geq v$, linked by the following ASP rules:

$$[x \geq L].$$
$$\forall v \in [L..U - 1], \ [x \geq v] \leftarrow [x \geq v + 1]$$
$$\leftarrow [x \geq U + 1]$$

The value of $x$ is then the largest value $v$ such that $[x \geq v]$ is true but $[x \geq v + 1]$ is false. Similarly, an lb-founded continuous variable can be thought of as an infinite set of ASP variables $[x \geq v]$ where $\forall v < v', [x \geq v] \leftarrow [x \geq v']$, and ub-founded variables are analogous, but with ASP variables representing $x \leq v$ instead. Of course, this grounding is precisely what we want to avoid.

### 6.2.2 BFASP Semantics

We now define the stable model semantics for a BFASP $P$ that is valid (which we define later). We first consider the case where we only have standard or lb-founded variables.

**Definition 6.4 (Rule reduct).** Given a rule $r = (c(y, x_1, \ldots, x_n), y)$ and an assignment $\theta$, the rule reduct of $r$ w.r.t. $\theta$ is written $r^\theta$ and is equal to the positive constraint that we get by replacing each $x_i$ where $x_i$ is not in $V_F$, or for which $c$ is not decreasing w.r.t. $x_i$, with $\theta(x_i)$.

**Definition 6.5 (CP reduct).** The reduct of a BFASP $P$ w.r.t. an assignment $\theta$ is written $P^\theta$ and is the following positive CP: $\{r^\theta : r \in \text{rules}(P), \text{trivial}(r^\theta)\}$. 
That is the reduct of the program is simply the collection of all its rule reducts that are non-trivial. Recall that non-trivial constraints are the ones that eliminate at least one value combination of values from the initial domains of their variables. Non-trivial constraints are essentially a generalization of tautologies over Boolean variables. The next theorem shows that the CP reduct generalizes the well-known Gelfond-Lifschitz reduct defined for normal logic programs.

**Theorem 6.1.** The CP reduct defined above generalizes the well-known Gelfond-Lifschitz (GL) reduct (Gelfond and Lifschitz, 1988a) used to transform normal logic programs to positive ASP programs, and is exactly equivalent to the GL reduct in the special case where P is a standard ASP program with normal rules over Boolean variables.

**Proof.** Consider a normal rule \( a ← p_1 \land \ldots \land p_n, \neg q_1 \land \ldots \land \neg q_m \). In BFASP format, this would be a rule constraint \((a ← p_1 \land \ldots \land p_n \land \neg q_1 \land \ldots \land \neg q_m, a)\). The constraint is increasing in \( a \) and the \( q_i \)'s, and is decreasing in the \( p_i \)'s. Given any solution \( \theta \), the GL reduct is constructed by removing all rules where one of the \( q_i \) is true in \( \theta \), and then removing all negative literals from all rules. Consider what happens when constructing the CP reduct. The constraint is not decreasing w.r.t. the \( q_i \)'s, so we substitute the value of \( \theta(q_i) \) for \( q_i \) for each \( i \) into the constraint. If any of the \( q_i \) is true in \( \theta \), then after substituting the values in, \( c \) becomes a tautology so we do not add it to \( P^\theta \), which is the same as removing the rule in the GL reduct. If all the \( q_i \)'s are false, then after substituting the values in, the constraint becomes \( a ← p_1 \land \ldots \land p_n \), which we add to the CP-reduct. This is again the same as in the GL reduct where we remove all negative literals. \( \square \)

We are now in a position to define what a stable assignment for a BFASP is.

**Definition 6.6 (Stable assignment).** An assignment \( \theta \) is a stable assignment of a BFASP \( P \) iff:

1. \( \theta \) satisfies \text{cons}(P), and

2. \( \theta =_{V^F} MA(P^\theta) \).

We can now extend the definition of CP reduct and stable model semantics to ub-founded variables as follows: for each ub-founded variable \( x \) in the program, create an lb-founded variable \( x' \) and replace every occurrence of \( x \) in \( P \) with \( -x' \), and then map
the stable models of the modified program to the original program via $\theta(x) = -\theta(x')$.
From this point on in this thesis, assume all founded variables are lb-founded, unless we explicitly specify otherwise.

**Example 6.2.** Consider the program $P$ with three founded integers variables $a, b, c$ and rules:

\[
\begin{align*}
(a &\geq -10, a) \quad (b \geq -10, b) \quad (c \geq -10, c) \\
(a &\geq b - c + 7, a) \quad (c \geq 5, c)
\end{align*}
\]

For the assignment $\theta = \{a = -8, b = -10, c = 5\}$, $P^\theta$ has the following positive constraints:

\[
\begin{align*}
a &\geq -10 \\
b &\geq -10 \\
c &\geq -10 \\
a &\geq b - 5 + 7 \\
c &\geq 5
\end{align*}
\]

The minimum model of $P^\theta$ is equal to $\theta$ modulo founded variables, and hence $\theta$ is a stable model of $P$. \hfill \Box

**Example 6.3.** Consider a BFASP with standard variable $s$, integer founded variables $a, b$, Boolean founded variables $x, y$, and the rules: $(a \geq 0, a), (b \geq 0, b), (a \geq b + s, a), (b \geq 8 \leftarrow x, b), (x \leftarrow \neg y \land (a \geq 5), x)$. Consider an assignment $\theta$ s.t. $\theta(x) = \text{true}, \theta(y) = \text{false}, \theta(b) = 8, \theta(s) = 9$ and $\theta(a) = 17$. The reduct of $\theta$ is the positive CP: $a \geq 0, b \geq 0, a \geq b + 9, b \geq 8 \leftarrow x, x \leftarrow a \geq 5$. The minimum solution that satisfies the reduct is equal to $\theta$, therefore, $\theta$ is a stable solution of the program. Consider another assignment $\theta'$ where all values are the same as in $\theta$, but $\theta'(s) = 3$. Then, $P^{\theta'}$ is the positive CP: $a \geq 0, b \geq 0, a \geq b + 3, b \geq 8 \leftarrow x, x \leftarrow a \geq 5$. The minimum solution that satisfies this positive CP is $M$ where $M(a) = 3, M(b) = 0, M(x) = M(y) = \text{false}$. Therefore, $\theta'$ is not a stable solution of the program. \hfill \Box

The next theorem shows that stable model semantics for BFASP generalize the stable model semantics for normal logic programs.

**Theorem 6.2.** For normal logic programs, the BFASP stable model semantics is equivalent to the stable model semantics of Gelfond and Lifschitz (1988a).
Proof. Follows from Proposition 6.1 and Theorem 6.1.

An important point in the formal description of BFASP is that for the sake of defining the semantics, it is easier to move the initial domains to the set of constraints instead of including them in the tuple that defines a BFASP. This is in contrast with the formal description of CSP in Section 2.2 where we explicitly made the initial domain $D$ part of the tuple that defined a CSP. For a founded variable $y$, there is ambiguity regarding semantics of a lower bound $l$ in the initial domain, i.e., whether it specifies the rule $(y \geq l, y)$ or the constraint $y \geq l$. We opt for the rule interpretation.\footnote{The upper bound for an lb-founded variable in the initial domain is unarguably a constraint.} In all subsequent discussion, if we do not mention any initial domain, assume it to be $[-\infty, \infty]$ and $\{false, true\}$ for numeric and Boolean variables respectively. As already illustrated in Section 6.1, initial domains can make monotonicity of a constraint (or rule) w.r.t. a variable more precise because a constraint that is non-monotonic w.r.t. one its variables might become monotonic in it once we consider an initial domain. Having stronger monotonicity relations expands the scope of BFASPs for which our semantics, solving algorithms, and language algorithms are described in this thesis.

6.3 Dependency Graph, Validity, And Stratification

The dependency graph of a BFASP $P$ is defined over founded variables. For each rule $r = (c, y)$, there is an edge from $y$ to all founded variable $x \in \text{vars}(c) \setminus \{y\}$. Each edge is marked increasing, decreasing, or non-monotonic, depending on whether $c$ is increasing, decreasing, or non-monotonic in $x$. The positive subgraph of a BFASP is the subgraph that we obtain by removing the edges marked increasing. In other words, the positive subgraph includes the edges marked decreasing or non-monotonic. We assign a unique number $id$ to every strongly connected component in the positive dependency graph and map every variable in the component to this number through a function $\text{scc}$ that preserves the topological order of the components. Next, we say that a BFASP is valid iff there is no edge marked non-monotonic within any SCC in its positive dependency graph. This is a broad class of programs that includes all normal logic and linear programs. Our stable
model semantics for BFASP guarantees that for any valid BFASP, stable models represent non-circular derivations.

Validity, as defined above, is essential to avoid circular or unstable solutions. This can be demonstrated by a simple program with two rules: \((a \geq |b|, a), (b \geq |a|, b)\). In the reduct, we substitute the non-monotonic arguments during the reduct, which means we can substitute any value that is in the domain of \(a\) and \(b\) and that would be a valid solution. But actually, this is not a stable solution in the sense that the values of \(a\) and \(b\) are only high because of one another. However, we allow non-monotonic dependencies between two different strongly connected components in the positive dependency graph since by construction, it is impossible to have positive circular derivation involving multiple SCCs. The next example shows the dependency graph for a simple BFASP.

**Example 6.4.** Consider a program with \(lb\)-founded integers variables \(a, b\) and rule constraints:
\[ r_1 = (a \geq |b|, a), r_2 = (b \geq 2a - 6, b)\]. The dependency graph (which is also equal to the positive dependency graph) has edges \((a, b)\) and \((b, a)\) due to \(r_1, r_2\) respectively. Since the constraint \(a \geq |b|\) is non-monotonic in \(b\), and \(a\) and \(b\) are in the same component, this is not a valid BFASP.

The next example shows how only changing the initial domain of a founded variable can affect validity of a BFASP.

**Example 6.5.** Consider a BFASP with founded variables \(x\) and \(y\) and the case when \(x\) and \(y\) are in the same SCC. If \(y \geq \sin(x)\) where \(x\) has initial domain \((-\infty, \infty)\), then it is not a valid BFASP since the \(\sin\) function is not monotonic over this domain. However, if \(y \geq \sin(x)\) where \(x\) has initial domain \([0, \pi/2]\), then the BFASP is valid.

Another important notion related to dependency graph is stratification. Stratification is a stronger notion than validity. We say that a program is stratified iff all the edges within any SCC in the dependency graph are strictly marked decreasing (as opposed to increasing or decreasing in validity). Following convention from logic programming (Przymusinski, 1988), we also refer to validity and stratification as local stratification and validity respectively, where the term local stresses that all variables and rules are ground.
We use this distinction in Chapter 8 when we discuss non-ground variables, rules, and constraints.

### 6.4 Rule Form Restriction

Besides the validity restriction, for this and the next two chapters, we also impose that all rules \((c, y)\) have the form \((y \geq f(\bar{x}), y)\). This restriction simplifies the presentation, but at the same time, eliminates some otherwise legitimate rule forms (e.g. \(y > f(\bar{x})\) where \(y \in \mathbb{R}\)). Where necessary, we describe in footnotes what happens if we remove this restriction.

It is extremely important to note that if \(c^i_{\bar{x}_i}\), then \(f^i_{\bar{x}_i}\) and similarly if \(c^i_{\bar{x}_i}\), then \(f^i_{\bar{x}_i}\).

### 6.5 Completion For BFASP\(^{17}\)

The Clark completion for normal logic programs is a set of clauses that encodes the value of a founded Boolean variable is equal to the disjunction of all its rules. Recall that in ASP, completion is sufficient to encode logic programs as propositional theories if the program is tight, i.e., does not have any positive recursion.

We can generalize the concept for BFASP. The generalization of disjunction is \(\max\). Given a BFASP \(P\), its completion, written \(\text{Comp}(P)\), is equal to the following set of constraints:

\[
\{ y = \max \{ f(\bar{x}) : (y \geq f(\bar{x}), y) \in \text{rules}(P) \} : y \in V_f \}
\]

**Example 6.6.** For the BFASP in Example 6.3, the completion contains the following constraints:

\(^{17}\)Since neither the BFASP semantics described in this Chapter nor the soundness and completeness of the algorithms that we describe in the next chapter depend on completion, we only define completion for programs where all rules follow the rule form restriction of Section 6.4.
where the function $I$ is defined as equal to the first argument when the second argument is true, and equal to $-\infty$ when it is false. Note that for $x$, we used max instead of the usual disjunction in completion of logic programs; the two are the same once we consider the order true $>$ false on Boolean variables.

### 6.6 Extensions Of ASP In BFASP

In practice, ASP languages such as GRINGO support some special constructs such as choice rules, cardinality constraints, weight constraints etc. In this section, we show how such constructs can be handled using BFASP.

#### 6.6.1 Choice Rules

A choice rule in ASP \:\{a_1, \ldots, a_n\} $\leftarrow$ $x$ essentially says that if $x$ is true, then variables $a_i$ need no justification to be true, i.e., they become standard. This can be encoded in BFASP as follows. We can introduce standard variables $b_1, \ldots, b_n$, and post the following rules for $i \in 1 \ldots n$ with heads $a_i$ and the rule constraints: $a_i \leftarrow b_i \wedge x$.

This means that whenever $x$ is true, then the value of $a_i$ is true whenever the standard variable $b_i$ is true. Let us represent this equivalent set of BFASP rules for the above ASP choice rule by the function $\text{choice}$.

#### 6.6.2 Aggregates

Let us define an auxiliary function, $b2i$ that takes in a founded Boolean variable $b$, creates a founded 0-1 integer variable $i$, posts a rule that makes both equivalent, and returns the founded integer variable. More formally:
6.6 Extensions Of ASP In BFASP

\[(b2i(b) = i) \Leftrightarrow (i \geq 1 \leftarrow b, i), (i \geq 0, i)\]

That is, \(b2i(b)\) is equal to the founded integer \(i\) where \(i\) has two rules: \((i \geq 0, i)\) and \((i \geq 1 \leftarrow b, i)\).

Furthermore, \(A\) represents an aggregate, either one of \textit{max}, \textit{min}, or \textit{sum}.

\textbf{Aggregation In Body}

In ASP:

\[x \leftarrow k \ A \{a_1 = v_1, ..., a_n = v_n\}\]

means that the body of the rule is equal to the truth value of the aggregate being greater than or equal to \(k\).

To get the BFASP equivalent, when \(A\) is a sum aggregate, we need the following rule in BFASP:

\[(x \leftarrow v_1 \times b2i(a_1) + \ldots + v_n \times b2i(a_n)) \geq k, x)\]

When \(A\) is minimum or maximum, then we have already looked at how that can be modeled with a combination of a normal rule and a cardinality rule in Section 2.5.2, both of which are supported in BFASP.

\textbf{Aggregation In Head}

ASP also supports aggregation in the head. For the ASP rule,

\[k \ A \{a_1 = v_1, ..., a_n = v_n\} \leftarrow x\]

the BFASP equivalent includes the set of rules given by the function \textit{choice}([\(a_1, ..., a_n\), \(x\)]) plus the following, depending on \(A\).

If \(A\) is a sum aggregate, then add the constraint: \(v_1 \times b2i(a_1) + \ldots + v_n \times b2i(a_n) \geq k \leftarrow x\)
If $A$ is minimum, then add the constraint $\neg x \lor \neg z$ and the rule $z \leftarrow \bigvee_{i \in 1..n, \bar{z}_i < k} a_i$.

If $A$ is maximum, then add the constraint $\neg x \lor z$ and the rule $z \leftarrow \bigvee_{i \in 1..n, \bar{z}_i \geq k} a_i$.

6.6.3 Discussion About BFASP Equivalent Constructs

The choice rule and aggregation in body and head are sufficient to cover the standardized ASP language as given by Calimeri et al. (2013). Although we have provided the BFASP equivalent constructs for these extensions of normal logic rules, it is debatable whether they are even needed in BFASP at all in practice. For example, it is not clear whether the conditional foundedness provided by choice rules and aggregation through head is ever needed in practice. Most often, the choice rules are used without a body, in which case they can be seen as a compensation for the lack of representation of standard variables in ASP.

6.7 Benchmarks

In this section, we define some benchmarks for BFASP. All the benchmarks include foundedness over numeric variables.

6.7.1 Shortest Path And Road Construction

We show how all-pairs shortest path program can be modelled in BFASP. We give the encoding for undirected graph, the encoding for directed graph is even simpler. Let $n$ be the number of nodes, and let $N$ be the set of nodes, i.e. $N = \{1 \ldots n\}$. Let $E$ be a set of weighted directed edges $(u, v, w)$ where $u, v,$ and $w$ represent the source, destination, and weight of the edge respectively.

Figure 6.1 shows the encoding of shortest path problem in BFASP. $sp_{x,y}$ is a ub-founded variable and represents the shortest path between nodes $x$ and $y$. The rules are similar to the ones that we described in Chapter 4 for the minimum connected dominating set with bounded diameter. Note that an undirected edge is treated as a pair of directed edges, and the rules $R_3$ and $R_4$ are written with that in mind.
\[ R_1 \quad \forall x, y \in N, x \neq y : \quad (sp_{x,y} \leq sp_{y,x}, sp_{x,y}) \]
\[ R_2 \quad \forall x \in N : \quad (sp_{x,x} \leq 0, sp_{x,x}) \]
\[ R_3 \quad \forall y \in N, (u, v, w) \in E : \quad (sp_{v,y} \leq w + sp_{u,y}, sp_{v,y}) \]
\[ R_4 \quad \forall y \in N, (u, v, w) \in E : \quad (sp_{u,y} \leq w + sp_{v,y}, sp_{u,y}) \]

Figure 6.1: BFASP encoding for the shortest path problem

Figure 6.2: A graph of roads and distances.

The following example illustrates why the above encoding, due to foundedness in BFASP, gives us the right solution to the shortest path problem, as opposed to the relational semantics in constraint solvers.

**Example 6.7.** For the graph shown in Figure 6.2, under the logical semantics used by standard CP solvers, an assignment such as \( \forall x, y, \theta(sp_{x,y}) = 0 \) is a solution to this \( P \), but does not give the shortest paths between the nodes. Under stable model semantics however, this is not a solution.

Calculating the minimum model of the reduct \( P^\theta \) gives \( sp_{1,2} = 8, sp_{1,3} = 11, sp_{1,4} = 5, sp_{2,3} = 7, sp_{2,4} = 13, sp_{3,4} = 6 \), which does not match \( \theta \), so \( \theta \) is not a stable model. It is easy to see that the only stable model for this problem is given by the minimum model of \( P^\theta \), which corresponds to the correct shortest path values.

Shortest path on its own is not a combinatorial problem, but there could be combinatorial problems whose description involves shortest path, as we discussed in Chapter 4. One such problem is road construction, where we are given a network of roads, and building each road is a Boolean decision variable. Furthermore, building a road incurs a cost, and we have a limited budget \( \text{budget} \), that is part of the problem description. Our objective is to minimize the sum of all shortest-path between any two nodes in the network.

Let us extend the problem description given above for the shortest path by associating
\[ R_1 \quad \forall x, y \in N, x \neq y : \quad (sp_{x,y} \leq sp_{y,x} \cdot sp_{x,y}) \]
\[ R_2 \quad \forall x \in N : \quad (sp_{x,x} \leq 0, sp_{x,x}) \]
\[ R_3 \quad \forall y \in N, (u, v, w) \in E : \quad (sp_{v,y} \leq w + sp_{u,y} \leftarrow build_{u,v}, sp_{v,y}) \]
\[ R_4 \quad \forall y \in N, (u, v, w) \in E : \quad (sp_{u,y} \leq w + sp_{v,y} \leftarrow build_{u,v}, sp_{u,y}) \]
\[ C_1 \quad \sum_{(u,v,w)\in E} \text{build}_{u,v} \times \text{cost}_{u,v} \leq \text{budget} \]
\[ O \quad \text{minimize} \quad \sum_{(x,y)\in N, x<y} sp_{x,y} \]

Figure 6.3: BFASP encoding for the road construction problem

a cost with each edge. In the model additionally, we have a Boolean decision variable build for each road which when true means that the edge is part of the graph. The shortest path rule for every edge is now predicated on building the road, and therefore we need to update it accordingly. Lastly, our objective function is to minimize the sum of all shortest path values. All this information is reflected in the encoding of road construction given in Figure 6.3.

### 6.7.2 Utilitarian Policies

Suppose a government wants to decide which policies to enact in order to maximize the happiness of its citizens. Enacting each policy incurs a certain cost. Additionally, the happiness of a citizen depends on the chosen policies as well as the happiness of other citizens. A citizen’s happiness can be increased (or decreased) by a certain amount if some other citizen’s happiness is above a certain threshold. More formally, the input consists a set of policies \( P \), a set of citizens \( C \), the cost of each policy \( p \) (\( \text{cost}_p \)), the total available budget \( (b) \), the utility of policy \( p \) for citizen \( c \) (\( \text{util}_{c,p} \)), the change in happiness of citizen \( c \) due to happiness of another citizen \( c' \) (\( \text{rel}_{c,c'} \)) and the threshold after which this change applies (\( t_{c,c'} \)). We can represent the happiness of a citizen \( c \) by the founded variable \( \text{hap}_c \), and the decision whether or not to enact a policy \( p \) by a Boolean variable \( \text{en}_p \). The objective is to maximize the sum of happiness of all citizens by choosing a set of policies to enact given a budget.
6.7 Benchmarks

\[
C_1 \quad \sum_{p \in P} en_p \times cost_p \leq b
\]

\[
R_1 \quad \forall c \in C : \quad (hap_c \geq \sum_{p \in P} util_{c,p} \times en_p + \sum_{c' \in C} rel_{c,c'} \times (hap_{c'} \geq t_{c,c'}, hap_c))
\]

\[
O \quad \text{maximize } \sum_{c \in C} hap_c
\]

Figure 6.4: BFASP encoding of Utilitarian Policies

The encoding is given in Figure 6.4. \(C_1\) is the budget constraint. \(R_1\) is the central rule in the program, and it says that the happiness of a citizen is greater than or equal to the sum of happiness that the citizen derives from the policies plus the sum of happiness from happiness of other citizens. The objective \(O\) is to maximize the overall happiness of all citizens.

6.7.3 Company Controls

Company Controls (Kemp and Stuckey, 1991, Mumick et al., 1990) is a widely cited problem in logic programming, especially in the context of aggregates. The goal of the problem is to minimize investments by a source company while ensuring that it controls a destination company. Control by a company \(x\) over another company \(y\) is exerted by either having a majority of shares in \(y\), or indirectly by having control over other companies whose cumulative shares in \(y\) plus \(x\)’s own shares exceed 50 percent of the total shares of \(y\). Let us formalize this.

Given a set of companies \(C\), and two companies \(s, t \in C\), we want company \(s\) to gain control of company \(t\) at minimum cost. Each company \(i\) has \(m_i\) shares in total. A company \(i\) can gain control of company \(j\) if it controls more than 50% of company \(j\)’s shares either directly (owns shares in \(j\) itself) or indirectly (controls a company that owns shares in \(j\)). Initially each company \(i \in C\) owns \(a_{i,j}\) of company \(j\)’s shares. Company \(s\) is going to try to gain control of company \(t\) by either buying shares in \(t\) directly, or by buying shares of other companies to gain control over them and then using them to buy more shares to gain control over additional companies, etc., until it controls \(t\). Each company \(i\) has a limited amount of cash \(budget_i\) to buy shares with. Let \(c_i\) represent whether company
$R_1 \quad (c_s, c_s)$

$R_2 \quad \forall j \in C : \quad (c_j \leftarrow \sum_{i \in C} c_i \times \left( a_{i,j} + b_{i,j} \right) > 0.5 \times m_j, c_i)$

$C_1 \quad c_t$

$C_2 \quad \forall i, j \in C : \quad c_i \lor b_{i,j} = 0$

$C_3 \quad \forall i \in C : \quad \sum_{j \in C} a_{i,j} + b_{i,j} \leq m_i$

$C_4 \quad \forall i \in C : \quad \sum_{j \in C} b_{i,j} \leq \text{budget}_i$

$O \quad \text{minimize } \sum_{j \in C} p_j \times b_{s,j}$

Figure 6.5: BFASP encoding of Company Controls

$s$ controls company $i$. Note that we assume company $s$ controls itself, so $c_s = \text{true}$ by default. Let $b_{i,j}$ be the amount of stocks of company $j$ that company $s$ forces company $i$ to buy. Note that company $s$ can only force company $i$ to buy if it controls company $i$, so $\neg c_i \rightarrow b_{i,j} = 0$. Let the stock of company $j$ have a fixed market price of $p_j$.

The BFASP model of company controls is given in 6.5. $R_1$ says that the source company $s$ controls itself by default. $R_2$ says that $s$ controls $j$ if it controls >50% shares in it. $C_1$ models that control over the target company $c_t$ must be true in the solution. $C_2$ constrains buying decisions to controlled companies only, and says that buying should be 0 in all other cases. $C_3$ is a capacity constraint that says that we cannot buy more than what is available. $C_4$ is the budget constraint and $O$ optimizes the total investment by the source company.

This version of company controls problem is different from the one used in ASP competitions where buying options are limited to a relatively small number of preset packages. Our version allows companies to buy shares in any increment. If the $m_i$’s are large, grounding our version to an ASP program will quickly cause the solver to run out of memory.

---

18https://www.mat.unical.it/aspcomp2011/FinalProblemDescriptions/CompanyControlsOptimize
6.8 Related Work

The formalism that is close to BFASP is Fuzzy Answer Set Programming (FASP) (Blondeel et al., 2013, Nieuwenborgh et al., 2006) (Section 2.6). The fuzzy atoms in FASP correspond to founded real variables in BFASP, and each logical connective in FASP can be expressed as a rule form in BFASP. Most importantly, from the implementation point of view, the MIP based unfounded set detection algorithm (Janssen et al., 2008) given for FASP only detects unfounded sets in a complete solution, which means that it cannot prune partial solutions that contain unfounded sets. Therefore, the algorithm has a shortcoming that is similar to the shortcoming of CMODELS (Lierler, 2005) in handling Boolean unfounded sets. Recently, FASP solving via translation to classical ASP has been suggested (Mushthofa et al., 2014, 2015). These solvers do not seem to remove the grounding bottleneck as the program is still solved by an ASP solver. The central idea comes from the result that checking satisfiability of an FASP (in which the domains of variables are infinite) can be done by checking satisfiability of finite \( k \)-valued fuzzy programs where \( k \) is exponentially bounded by the number of variables in the program. The presented approach iteratively increases the value of \( k \) and in each iteration solves a propositional logic program, whose size can be exponential in \( k \) (and hence susceptible to grounding bottleneck). If an answer set of this translated program is found, then that corresponds to a fuzzy answer set of the original program. The presented approach has only been shown to work for finding satisfying assignments, and it is not obvious how the procedure can be used to solve our optimization benchmarks without running into performance issues.

Fuzzy ASP can be converted into BFASP rules with the same stable model semantics. In FASP, the model must specify how each logical connective acts on the value of its arguments. However, all the commonly used options allow the FASP to be directly translated into BFASP. For example, depending on whether we use the Lukasiewicz, Gödel-Dummett, or product norm, a FASP rule: \( x \leftarrow p_1 \otimes \ldots \otimes p_k \otimes \neg n_1 \otimes \ldots \otimes \neg n_m \) translates into one of the BFASP rules given in Figure 6.6.

It may initially appear that BFASP is very similar to FASP. However, BFASP is a significant generalization. It is far more generic in terms of what the head variable can represent (not just fuzzy Booleans but arbitrary numeric values) and it allows a wider range
of constraints in the rules (not just logical connectives, but also globals like \texttt{alldifferent}). Furthermore, the implementation that we describe in the next chapter can handle non-linear rules such as those produced by the product norm (which the MIP based method of (Janssen et al., 2008) cannot handle), and we have an unfounded set detection algorithm for BFASP that allows us to prune unstable partial models during search (which the MIP-based method of (Janssen et al., 2008) also cannot do).

There is another body of work in \textit{partial order programming} (Osorio and Jayaraman, 1996) that is similar to BFASPs. In this work, the authors also look at logic programs with rules of form $y \geq f(expr) \leftarrow \text{conditions}$. However, they only define the semantics of a limited subclass of such programs. The cases they consider, in our terminology, are as follows. Case 1, they consider any variable in \texttt{expr} or \texttt{conditions} as standard variables only and apply the standard logical semantics (Section 4.1 (Osorio and Jayaraman, 1996)). This is completely different from the stable model semantics and will give the wrong answers for a simple shortest path program. Case 2, they consider only \textit{hierarchical} programs where every variable in \texttt{expr} or \texttt{conditions} belongs to a previous SCC (Prop 5.1 of (Osorio and Jayaraman, 1996)). This is insufficient to model shortest path, as all those variables are in the same SCC. Case 3, they allow non-hierarchical programs, but same SCC variables can only interact through increasing monotonic functions, i.e., $f$ is increasing in all arguments that are in the same SCC as $y$. This allows shortest path to be modelled, but is still far less expressive than the stable model semantics defined in this chapter. The semantics that we shall describe for BFASP allows $f$ to be decreasing in arguments from the same hierarchical level, and thus we shall truly generalize negation by failure to continuous domains, giving the generalization of stable model semantics to continuous variables.

Figure 6.6: Translation of Fuzzy ASP to BFASP
Chapter 7
BFASP Implementation

This chapter describes an algorithm to implement BFASP semantics. The algorithm is based on unfounded set detection during the search. We begin by introducing some concepts that help us in the analysis of our algorithm. We then provide a brief sketch of the algorithm, which is followed by the definition of unfounded sets for BFASPs. Using the definitions and results for unfounded sets, we then prove that our algorithm implements the reduct-based stable model semantics for BFASP defined in Chapter 6. After the analysis, we describe the algorithm in detail and provide an experimental evaluation on some benchmarks to demonstrate its strength in solving problems involving founded numeric variables.

7.1 Strongest Bound And Stabilizers

Definition 7.1 (Strongest bound). Given a constraint \( c(y, \bar{x}) \) s.t. \( c_y \) and a domain \( D \), the strongest bound on \( y \) due to \( c \) w.r.t. \( D \), written \( sb(c, y, D) \), is equal to:

\[
\max\{\min\{y_i \mid (y_i, \bar{x}_i) \in c\} : \bar{x}_i \in D(\bar{x})\}
\]

If \( c \equiv y \geq f(\bar{x}) \), then \( sb(c, y, D) = \max\{f(\bar{x}) : \bar{x}_i \in D(\bar{x})\} \).

Intuitively, the strongest bound provides us with the most optimistic lower bound on \( y \) due to a constraint given a domain. For a rule \( r = (c,y) \), we overload the arguments and define the strongest bound \( sb(r, D) \) as the strongest bound of \( c \) on the head, i.e., \( sb(r, D) = sb(c, y, D) \).

To implement the algorithm that we present shortly, we focus on rules for whose constraints we can compute the strongest bound efficiently, given the domain. An even more specific case is where we can just look at the maximum and minimum values in the
domains of \( \bar{x} \) and compute the strongest bound on \( y \). This is similar to how constraint solvers perform bounds propagation. Of course, this is only possible if \( f \) is monotonic in all variables in \( \bar{x} \). In that case, plugging in \( \max(D, x_i) (\min(D, x_i)) \) if \( f \) is increasing (decreasing) in \( x_i \) and evaluating the resulting function gives us the strongest bound on \( y \).\(^{19}\)

But valid BFASPs, as discussed in Section 6.3, can also have non-monotonic arguments from other SCCs, in which case looking at the boundaries of domains of variables in \( \bar{x} \) might not be sufficient to give us the strongest bound on \( y \), and we would need more sophisticated reasoning capabilities to compute it. However, when all variables in \( \bar{x} \) are fixed, then the strongest bound is simply equal to what we get by plugging in the single values in domains of variables in \( \bar{x} \).\(^{20}\)

Carrying this motivation further, we introduce a stabilizer for each rule whose job is to compute an approximation of the strongest bound function when all variables in \( \bar{x} \) are not fixed, but exactly the strongest bound function when they are fixed. The following definition formalizes this.

**Definition 7.2 (Stabilizer).** A stabilizer for a rule \( r \), is a function that takes in a domain \( D \), is written \( sjb(r, D) \), and outputs \( k \) s.t. \( k \geq sb(r, D) \) when \( D \) is not an assignment, and \( k = sb(r, D) \) when it is.\(^{21}\)

The symbol \( sjb \) stands for strongest justified bound. Note that this is different from strongest bound. More particularly, strongest justified bound is what a stabilizer works out as an attempt to compute (approximate) the strongest bound efficiently, given the current decisions. To simplify the notation, whenever it is clear from context that we are talking about domain \( D \), we remove it from the list of arguments.

\(^{19}\)We remind the reader of the important caveat mentioned in Chapter 6 that if \( c\uparrow^{x_i} \), then \( f\downarrow^{x_i} \) and similarly if \( c\downarrow^{x_i} \), then \( f\uparrow^{x_i} \).

\(^{20}\)We could generalize this restriction to the case where we only require the non-monotonic values to be fixed, but we do not do so for simplicity.

\(^{21}\)Note that it might be possible to apply the bounds reasoning to compute the strongest justified bound using a stabilizer even when we remove the rule form restriction of Section 6.4. First, reverse the increasing/decreasing sign described above since we have now switched from \( y \geq f(\bar{x}) \) to \( c(y, \bar{x}) \). Second, the strongest bound is equal to the minimum value of \( y \) that satisfies it, therefore, a good stabilizer should aim at a value as close to this minimum value as possible when the domain is not fixed. Of course, for correctness, when the domain is fixed, then the strongest justified bound must be equal to this value. For example, consider the rule \( (\max(y, x) \geq a, y) \) then using the bounds reasoning, for \( a \) and \( x \) we substitute \( a' = \max(D, a) \) and \( x' = \min(D, x) \) respectively. A stabilizer could return \( a' \) if \( a' > x' \), and \(-\infty \) otherwise (this is actually the strongest bound as well). Essentially, we are implementing the logic of the equivalent rule \( (y \geq a \leftarrow a > x) \) that does follow the rule restriction of Section 6.4.
In case when $f$ is monotonic in all arguments in $\bar{x}$, e.g. when $f$ is a linear function, product, conjunction, disjunction, minimum, or maximum, then we have already described the implementation of efficient stabilizers that work just by looking at the boundaries of domains of variables in $\bar{x}$. Even for the non-monotonic case, there can still be efficient techniques to compute $sjb$ reasonably. For example, in a rule like $y \geq 5 \times \sin(x)$, until $x$ is fixed, we could return the 5 as the strongest justified bound, instead of returning $\infty$. An important point is that constraint propagator for $y \leq f(\bar{x})$ carries out the same reasoning to compute the upper bound of $y$, and we could leverage them to compute the strongest justified bound on $y$ for the rule $(y \geq f(\bar{x}), y)$.

7.2 Overview And Analysis Of The Algorithm

The algorithm that we describe in the next section is essentially an extension of the source pointer technique that we have already discussed in great length in Section 3.4, along with a class of unfounded set algorithms that build on the idea. Our algorithm is also very similar, with the main difference being that we carry out the reasoning on bounds of variables, instead of their truth values (which is a special case). Another critical difference is that ASP solvers prune off all unfounded sets w.r.t. a partial assignment (non-singleton domain), while we only guarantee this for complete assignments (singleton domains), when all stabilizers return the strongest bound as $sjb$. In the special case where all arguments in all rules are monotonic and we get the tightest possible strongest justified bound from every rule, our algorithm also prunes all unfounded sets (which we have yet to define) w.r.t. the current domain.

7.2.1 Justified Bounds

The nodes in the justification graph of our algorithm are bounds of the form $[y \geq k]$. In the algorithm, a justified bound represents what is already justified given the current decisions. A set of such bounds $J$ intuitively represents a justified domain $[-\infty, jb(J, y)]$ for a variable $y$ where the justified bound $jb(J, y)$ is equal to $\max\{k : [y \geq k] \in J\}$. Let us give $\text{Recall that interval notation } [l, u] \text{ means the set of values } \{x : l \leq x \leq u\}.$
this interchangeable representation a name, and define it as the function $\text{boundsToDom}(J)$, such that for every $y \in V_J$, $\text{boundsToDom}(J)(y) = [-\infty, jb(J, y)]$.

The algorithm is implemented as a set of propagators, one for each SCC in the dependency graph that are prioritized in the topological order such that all SCCs on which an SCC depends run before it. Broadly, each propagator works as follows. Given a domain $D$ and the nodes in justification graph $J$, it uses the intersection of $D$ and $D_J = \text{boundsToDom}(J)$, in the computation of strongest justified bounds of rules. The $sjb$ of each rule in turn returns a bound, which the algorithm adds to $J$, and it does so, until a fixed point is reached, that is, when all $sjb$ return a value that is less than or equal to the maximum value of head in the domain $D \cap D_J$. The algorithm then propagates $D = D \cap D_J$, and also posts an explanation which specifies that given the current domain, the founded variables can only go so high. The explanation is a generalization of the loop nogoods that are used in conflict driven answer set solvers to explain pruning off unfounded sets.

Note that there is no need to actively maintain $D \cap \text{boundsToDom}(J)$ and we can achieve the desired effect in a simpler way. First, consider only the rules $(y \geq f(\bar{x}))$ where all arguments are monotonic. We describe how to return $sjb(r, D \cap \text{boundsToDom}(J))$ while working directly with $D$ and $J$. We simply return $f(x'_1, ..., x'_n)$, where for each $x_i$ for which $f^U_{x_i} = \min(ub(D, x_i), jb(J, x_i))$ and for each $x_i$ for which $f^L_{x_i} = lb(D, x_i)$. Regarding rules containing non-monotonic arguments, since our algorithm posts a separator propagator for each SCC, it follows that for a valid BFASP, the domains of all non-monotonic arguments will be completely justified (as determined by running a fixed point loop on $sjb$ of rules of that SCC to which the non-monotonic argument belongs). Therefore, we leave out the special treatment of non-monotonic arguments.

To avoid the discussion of $\text{boundsToDom}$ in the subsequent text, we work directly with $D$ and $J$. This means that we update the arguments list of $sjb$ to include $J$. To do this, we define $sjb(r, D, J) = s jb(r, D \cap \text{boundsToDom}(J))$ (and similarly for $sb(r, D, J)$). Whenever clear from the context, we drop the last two arguments for simplicity in notation.
7.2 Overview And Analysis Of The Algorithm

7.2.2 Unfounded Sets And Correctness Results

Let us now turn our attention to defining unfounded set of bounds for BFASPs. Before we do that, we introduce one definition. Given a set of bounds \( Y \) of the form \([y \geq k]\), let the saturation of \( Y \) from below be written as \( sat^{-}(Y) \) and be defined as \( Y \) plus addition of all redundant bounds \([y \geq k']\) such that \( k' < k \) and \([y \geq k] \in Y \).

Similarly, define \( sat^{+}(Y) = \{ [y \geq k'] : [y \geq k] \in Y, k' \geq k \} \). Additionally, let \( B \) be a set of all bounds of the form \([y \geq k]\) where \( y \in V \). It is intuitive to think of \( B \) as a universal set of greater than or equal to bounds. In all subsequent discussion in the analysis of our algorithm, we use \( B \) to represent the set of all bounds like we just described. For a set of such bounds \( U \), \( B \setminus sat^{+}(U) \) means that for any variable in \( U \), we take its lowest value in \( U \), and subtract all values greater than or equal to that from \( B \). With this definition, we are ready to define unfounded sets.

**Definition 7.3.** Let \( U \) be a set of bounds of the form \([y \geq k]\). Let \( J = B \setminus sat^{+}(U) \). We say \( U \) is an unfounded set w.r.t. domain \( D \) iff for all rules \( r \):

\[
\text{sb}(r, D, J) \leq \text{jb}(J, \text{head}(r)).
\]

Intuitively, this means that if \( U \) is unfounded, then given only the bounds not in \( U \), no rule can ever justify a bound in \( U \), or in other words, all bounds in \( U \) are either underivable or only have circular derivations involving other bounds in \( U \).

The following results establishes the relation between stable assignments and unfounded sets. It says that an assignment is stable iff there is no unfounded set w.r.t. the assignment, that the assignment intersects with.

**Theorem 7.1.** Given a full assignment \( \theta \) that satisfies all rules and constraints in \( P \), \( \theta \) is a stable assignment \( \iff \) there does not exist \( U \) which is unfounded w.r.t. \( \theta \) such that \( \theta \) satisfies at least one bound in \( sat^{+}(U) \).

**Proof.** (\( \Rightarrow \)): We show that if \( \theta(y) = v \) and \([y \geq v]\) is in some unfounded set \( U \), then \( \theta \) is not a stable assignment. By definition of unfounded set, for all rules \( r \) with \( y \) as head, there exists \( v' \) with \( v' < v \) s.t. all those rules are satisfied with \( y = v' \). We construct \( \theta' \) where
$y = v'$ instead of $v$. Then $\theta'$ is a solution of $P^\theta$, which means that $\theta$ is not a minimum model of $P^\theta$ and hence is not a stable assignment of $P$.

(⇐): Suppose $\theta$ is not a stable assignment. Let $\theta'$ be the least model of $P^\theta$. Let $U$ be $\{[x \geq v] : v > \theta'(x)\}$. Then $U$ is an unfounded set w.r.t. $\theta$. By definition of least models, there must exist $y$ such that $\theta(y) > \theta'(y)$. Then $y = \theta(y)$ satisfies a bound in an unfounded set.

The following lemma says that an unfounded set w.r.t. a domain is also unfounded w.r.t. all its contractions.

**Lemma 7.1.** If $U$ is unfounded w.r.t. $D$, and $D' \subseteq D$, then $U$ is unfounded w.r.t. $D'$.

**Proof.** From the definition of strongest bound function, it follows that for every rule $r = (c, y)$, $sb(c, y, D') \leq sb(c, y, D)$ and since $U$ is an unfounded set w.r.t. $D$, we have that $sb(c, y, D) \leq jb(B \setminus U)$, which means that $sb(c, y, D') \leq jb(B \setminus U)$. Therefore, $U$ is also an unfounded set w.r.t. $D'$.

The next theorem, which follows from the previous results, establishes that an unfounded set w.r.t. a domain, cannot be part of any stable assignment which is a contraction of that domain.

**Theorem 7.2.** Given a domain $D$, if $U$ is unfounded w.r.t. $D$, then no stable assignment of $P$ that is a contraction of $D$ can satisfy any bound in $U$.

**Proof.** If $\theta \subseteq D$ and satisfies a bound in some unfounded set $U$, then by Lemma 7.1, $U$ is also an unfounded set for the domain $\theta$, and then by Theorem 7.1, $\theta$ is not a stable assignment.

With the definition of unfounded sets and its relation with stable model semantics, we now show that our algorithm, whose sketch we described, correctly implements stable model assignments by pruning off unfounded sets during the search.

The first result says that if we apply the stabilizers until a fixed point is reached, then all the bounds that are not in $J$ form an unfounded set.
Theorem 7.3. Given a domain $D$, if we start with $J = \emptyset$ and apply the operators $srb(r, D, J)$ for all rules $r$ until $J$ reaches a fixed point, then $U = B \setminus \text{sat}^-(J)$ is an unfounded set w.r.t. $D$.

Proof. Since we are at fixed point, $srb(r, D, J) \leq jb(J, \text{head}(r))$ for all rules $r$. Also, for all rules $r$, $sb(r, D, J) \leq srb(r, D, J)$, which means that $U$ is an unfounded set by definition. \hfill \Box

The next theorem shows that if we reach a full assignment using our algorithm and there are no unfounded sets w.r.t. that assignment, then that assignment is a stable one.

Theorem 7.4. Given an assignment $\theta$, if the fixed point calculation from Theorem 7.3 produces a $U$ s.t. none of the assignments in $\theta$ satisfy a bound in $U$, then $\theta$ is a stable assignment.

Proof. By construction, each bound that is added to $J$ can be derived using some rule and the bounds previously in $J$. Therefore, none of them can belong to any unfounded set. If $\theta$ does not satisfy any bound in $U$, then for all $x \in \mathcal{V}_F, [x \geq \theta(x)]$ must be in $J$ and cannot be in any unfounded set. Therefore, by Theorem 7.1, $\theta$ is a stable assignment. \hfill \Box

7.3 Algorithm

In this section, we explain our algorithm in detail.

Let $\text{body}(r) = \{x_1, \ldots, x_n\}$, and $\text{body}^+(r)$ and $\text{body}^-(r)$ be the subsets of $\{x \mid x \in \text{body}(r), \text{scc}(x) = \text{scc}(y)\}$ in which $f_r$ is monotonically increasing and decreasing respectively. Let $\text{body}^0(r) = \{x \mid x \in \text{body}(r), \text{scc}(x) \neq \text{scc}(y)\}$. Let $\text{rules}(y) = \{r \mid \text{head}(r) = y\}$. Let $\text{int}_\text{rules}(y) = \{r \mid \text{head}(r) = y, \text{body}^+(r) \neq \emptyset\}$. Let $\text{ext}_\text{rules}(y) = \text{rules}(y) \setminus \text{int}_\text{rules}(y)$. We also initialize a static data structure $\text{EventToRule}$ consisting of pairs $(e, r)$ where $e$ is a domain change event that might cause $r$ to stop justifying a bound on $y$. That is, for each $x_i$ we add $(\text{ub}_\text{event}(x_i), r), (\text{lb}_\text{event}(x_i), r)$ or $(\text{change}_\text{event}(x_i), r)$ depending on whether $f_r$ is increasing in $x_i$, decreasing $x_i$, or neither respectively.

For convenience, we extend the justification graph such that it allows us to access all incoming edges from a node. The justification graph $\text{JustGraph}$ now consists of a set of triples $(b, r, s)$, where $b$ is a bound that is justified by some rule constraint $r$, and $s$ is a set of other bounds with variables in the same level whose justification was used to infer
that $b$ might be justified. We define several functions of the current justification graph $\text{JustGraph}$ and the current domain $D$. For brevity, we leave out the implicit arguments of $\text{JustGraph}$ and $D$. Let $jb(x) = \max\{v \mid \exists r, \exists s, ([x \geq v], r, s) \in \text{JustGraph}\}$. Let $ext\_jb(x) = \max\{v \mid \exists r \in \text{ext\_rules}(x), \exists s, ([x \geq v], r, s) \in \text{JustGraph}\}$.

To allow nogood learning, we need to explain unfounded sets, so we need a function $\text{expl}(r)$ that returns a set of literals that explain why no bound stronger than $[y \geq sjb(r)]$ can be justified. For the naive implementation of $sjb(r)$ above, when not all vars in $\text{body}^0(r)$ are fixed, we return $\text{expl}(r) = \emptyset$. Otherwise, we return $\text{expl}(r) = \{[x \leq ub(x)] \mid x \in \text{body}^+ (r)\} \cup \{[x \geq lb(x)] \mid x \in \text{body}^- (r)\} \cup \{[x = val(x)] \mid x \in \text{body}^0 (r)\}$.

We post a foundedness propagator for each (non-empty) SCC, which is responsible for the variables in the component. The priority of each propagator, as explained previously, is the same as the $id$ of that component. The propagators support four methods: $\text{propagate}()$, $\text{processEvents}()$, $\text{getUnfoundedSet}()$ and $\text{getExpl}()$. The pseudo-code for each of these functions is given in Figures 7.1, 7.2, 7.3, and 7.4 respectively. The function $\text{propagate}()$ first calls $\text{processEvents}()$ to process any domain change events and dejustify any bounds whose justification has been made invalid by the domain changes. Next, if there is any bound in the current domains of any variable $x$ with $\text{sc}(x) = \text{id}$ that is not justified, it calls $\text{getUnfoundedSet}()$ to either rejustify the bounds, or find an unfounded set. If an unfounded set is found, then it sets the bounds in the unfounded set false with the explanation from the call $\text{getExpl}()$. If there are any unprocessed variables left, the propagator requeues itself in the propagation queue so that higher priority propagators can run before this propagator is run again.

The function $\text{processEvents}()$ crawls through the existing justification graph to find out what bounds lose their justification due to new domain changes. The function $\text{getUnfoundedSet}()$ does a fix-point calculation using the $sjb()$ to find out what bounds can be rejustified. The function $\text{getExpl(Unfounded)}$ resolves the explanations for why each bound cannot be justified in order to derive an explanation for the whole unfounded set. See the pseudocode for more details.

The algorithm is initialized as follows. At the root node, $\text{JustGraph} = \emptyset$. We wake up all foundedness propagators. On the first call to each foundedness propagator at the
7.3 Algorithm 117

root node, we initialize LostExt to all the variables in that level instead of the usual ∅. JustGraph is a trailed data structure, so upon backtracking, it is reverted to its previous value.

propagate()

processEvents()

HasUnjust = \{ y | scc(y) = id, jb(y) < max(D(y)) \}

if (HasUnjust ≠ ∅)

Unfounded = getUnfoundedSet()

if (Unfounded ≠ ∅)

E = getExpl(Unfounded)

for (y ∈ Unfounded)

Propagate y ≤ jb(y) with explanation E

if (HasUnjust ≠ ∅) requeuePropagator()

Figure 7.1: Main propagation algorithm

Let us demonstrate how the above algorithm works on the benchmark road construction. For better readability, only for this example, we bring back ub-founded variables to represent shortest paths.

Example 7.1. Recall the road construction encoding as given in Figure 6.3. Also recall the instance from Example 6.7, but now we add the cost to build each road and the available budget: cost_{1,2} = 50, cost_{2,3} = 30, cost_{3,4} = 20, cost_{4,1} = 60 and budget = 100.

Since the build variables do not depend on anything else, the SCC algorithm will set them to level 0. The set of sp variables are cyclically dependent on each other, so the SCC algorithm will set them to level 1. The first set of rules in the model force sp_{x,y} = sp_{y,x} and we will not mention them any further. At the root node, the second set of rules give [sp_{x,x} ≤ 0] for all x with empty justifications. Depending on the evaluation order, different rules in the third set may justify the same bound. We will just pick any one of them in this example. The third set of rules give: [sp_{1,2} ≤ 8], [sp_{2,3} ≤ 7], [sp_{3,4} ≤ 6], [sp_{1,4} ≤ 5] with empty justifications, [sp_{1,3} ≤ 11] with justification [sp_{3,4} ≤ 6], and [sp_{2,4} ≤ 13] with justification [sp_{3,4} ≤ 6]. No stronger upper bounds can be justified, thus the foundedness propagator forces [sp_{1,2} ≥ 8], [sp_{2,3} ≥ 7], [sp_{3,4} ≥ 6], [sp_{4,1} ≥ 5], [sp_{1,3} ≥ 11], [sp_{2,4} ≥ 13] with explanation ∅. Note that what unfounded
processEvents()

Dejustified = ∅
LostExt = ∅

for (e ∈ DomEvents)
  for (r s.t. (e, r) ∈ EventToRule)
    for (y ≥ k, r′, s) ∈ JustGraph s.t. r = r′
      if (k = ext jb(y)) LostExt = LostExt ∪ {y}
      JustGraph = JustGraph − {((y ≥ k), r′, s)}
      Dejustified = Dejustified ∪ {y ≥ k}

for (y ∈ LostExt)
  r = argmax_{r′ ∈ ext rule(y)} sjb(r′)
  JustGraph = JustGraph ∪ {((y ≥ sjb(r)), r, ∅)}
  Dejustified = Dejustified − {y ≥ sjb(r)}

while (∃b ∈ Dejustified)
  Dejustified = Dejustified − {b}
  for ((bound, rule, supports) ∈ JustGraph s.t. b ∈ supports)
    JustGraph = JustGraph − {((bound, rule, supports)}
    Dejustified = Dejustified ∪ {bound}

Figure 7.2: Processing domain change events

set detection is doing is basically to calculate the most optimistic values of the shortest paths given the current set of decisions, and then set those values as the lower bounds of the sp_{x,y}.

Suppose search tries b_1 = true. Propagation forces sp_{1,2} = 8. The budget constraint forces b_4 = false. This domain change event triggers the rules sp_{4,1} ≤ 5 ← b_4 and sp_{1,3} ≤ 5 + sp_{3,4} ← b_4 to stop justifying bounds on their heads, so [sp_{4,1} ≤ 5] and [sp_{1,3} ≤ 11] become dejustified. The third set of constraints now allow us to rejustify a bound of [sp_{1,3} ≤ 15] with justification [sp_{2,3} ≤ 7] followed by [sp_{4,1} ≤ 21] with justification [sp_{1,3} ≤ 15]. No stronger bounds can be justified, thus the foundedness propagator forces [sp_{1,3} ≥ 15] and [sp_{4,1} ≥ 21] with explanation {¬b_4}.

Suppose then search tries b_2 = true. Propagation forces sp_{2,3} = 7 and sp_{1,3} = 15. Suppose search then tries b_3 = true. Propagation forces sp_{3,4} = 6, sp_{2,4} = 13 and sp_{4,1} = 21. This gives a stable model with objective value: 6 + 7 + 8 + 13 + 15 + 21 = 70. Suppose we backtracked and tried b_3 = false instead. This domain changed event causes [sp_{3,4} ≤ 6] and [sp_{4,1} ≤ 21] to become dejustified. Since [sp_{3,4} ≤ 6] was used to support [sp_{2,4} ≤ 13], that also becomes dejustified. No bounds can be rejustified, thus the foundedness propagator forces [sp_{2,4} ≥ ∞], [sp_{3,4} ≥ ∞], and
getUnfoundedSet()
    let z be such that z ∈ HasUnjust
    HasUnjust = HasUnjust − {z}
    Unfounded = Unfounded ∪ {z}
    UnprocRules = int.rules(z)
    while (∃r ∈ UnprocRules)
        UnprocRules = UnprocRules − {r}
        y = head(r)
        nb = min(sjb(r), max(D(y)))
        if (nb > jb(y))
            JustGraph = JustGraph ∪
            {{y ≥ nb}, r, [{x ≥ jb(x)| x ∈ body+(r)}]}
            UnprocRules = UnprocRules ∪
            {r′ | r′ ∈ R, y ∈ body+(r′), head(r′) ∈ Unfounded}
        if (nb = max(D(y)))
            Unfounded = Unfounded − {y}
    else
        for (l ∈ expl(r))
            if (D | l)
                HasUnjust = HasUnjust − {var(l)}
                Unfounded = Unfounded ∪ {var(l)}
                UnprocRules = UnprocRules ∪ int.rules(var(l))
    return Unfounded

Figure 7.3: Main unfounded set calculation

[sp_{4,4} ≥ ∞], with explanation \{-b_4, \neg b_3\}, i.e., there is no path from a, b or c to d. This stable model has an objective value of ∞.

7.4 Experimental Results

We compare our implementation with state of the art ASP system GRINGO (version 3.0.4) + CLASP (version 2.0.6), which we call CL+GR in this section. We refer to our implementation as CHUFFED. At present it only supports bound founded integers and Booleans. All experiments were run on a Lenovo model 3000 G530 notebook with a 2.1 GHz Core 2 Duo T6500 CPU and 3 GB of memory running Ubuntu 12.04. Every instance was run 10 times, and each timing (that is not —) in the tables in this section is the median of these 10 times. All times are given in seconds. For all experiments, the timeout for grounding and
getExpl(Unfounded)
  \[ E = \emptyset \]
  for \( y \in \text{Unfounded} \)
    for \( r \in \text{rules}(y) \)
      \[ E = E \cup \text{expl}(r) \]
  \[ E = E - \{ [y \leq v] \mid y \in \text{Unfounded}, v \geq jb(y) \} \]
  return \( E \)

Figure 7.4: Explaining unfounded sets

flattening was 5 minutes, and timeout for solving was 10 minutes.\(^{23}\) ShortPath: Finding the shortest path from a source node to a destination node in a directed graph of \( N \) nodes and \( E \) edges. See Section 6.7.1 for more details.

RoadCon: The road construction problem as defined in Section 6.7.1, on an undirected graph of \( N \) nodes, where \( D \) is the maximum distance between two points and \( C \) is the maximum cost to build a road segment.

CompCon: Company controls problem as defined in the introduction where \( C \) is the number of companies and \( S \) is the number of shares each company has. See 6.7.3 for further details.

UtilPol: Utilitarian problem as described in Section 6.7.2.

We chose these problems since they contain founded bounds on integers. The results from our experiments are presented in Tables 7.1 to 7.4 in the following format: each row represents the results for a particular instance. The performance of each system is given by several columns. The first column gives the grounding (gr) or flattening (flat) time. If the system failed to complete this phase, then a — is put in all its columns. The second column gives the best value produced by the system (omitted in Table 7.1 which deals with satisfiability). The third column tells whether the system provably found the optimal value. If a system did not produce any answer for an instance, then a — is put for this field. The final column gives the actual time that the solver took to solve the instance, the value is 600 if the solver did not complete before timeout or ran out of memory.

Table 7.1 presents comparison of CL+GR, a CP solver CPX and CHUFFED on the short-

\(^{23}\) All instances and encodings used are available in GRINGO and MINIZINC format at: http://ww2.cs.mu.oz.au/~pjs/bound_founded/
7.4 Experimental Results

The CP encoding views the shortest path problem as a combinatorial problem, where the solver constructs the shortest path by choosing a set of edges. CL+GR only produces a solution for the smallest instance, and fails to ground the other instances in the allocated time. The comparison between CHUFFED and CPX is more interesting. On smaller instances, performance of both systems is comparable but on bigger instances, the flattening plus the solving time of the combinatorial encoding increases significantly. We do not compare with CPX in the remaining experiments since it requires explicitly modelling the cyclic dependencies inductively which quickly blows up.

Table 7.2 compares the performance of CL+GR and CHUFFED on the road construction RoadCon benchmark. CL+GR starts to break down on the second instance but CHUFFED continues to give answers for instances that contain as many as 60 nodes.

Table 7.3 presents results on the company controls benchmark CompCon. Grounding is not the bottleneck for this benchmark, but the solving time of CLASP increases significantly as the number of stocks and number of companies are increased. For CHUFFED, increasing the number of companies makes the problem significantly harder but the effect of increase in the number of stocks is relatively mild as compared to CLASP. CHUFFED solves an instance that has 30 companies and fails on an instance in which there are 50 companies.

Table 7.4 presents results for utilitarian policies problem UtilPol. As in previous benchmarks, CHUFFED comprehensively outperforms CL+GR. The domains of rel and util variables were set to 0 . . . 5 in all instances. Any increase in the sizes of domains of these variables has no effect on the performance of CHUFFED but has a significant effect on the performance of CL+GR. The right of Table 7.4 shows the sizes of the grounded program and the running times of CLASP as the domains of variables are scaled by the number given. For a tenfold increase in the domain size, the size of the ground program increases from 748 kilobytes to 337 megabytes. The running time also increases significantly until timing out on the last 3 instances. In contrast CHUFFED requires 3.6kB and less than 0.005 seconds for all instances.
### Table 7.1: ShortPath

<table>
<thead>
<tr>
<th></th>
<th>CL+GR</th>
<th>CHUFFED</th>
<th>CPX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gr comp time</td>
<td>flat comp time</td>
<td>flat comp time</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>6.00</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.01</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.03</td>
<td>Y</td>
</tr>
<tr>
<td>50</td>
<td>20</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.05</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.37</td>
<td>Y</td>
</tr>
<tr>
<td>100</td>
<td>30</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.25</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.12</td>
<td>N</td>
</tr>
<tr>
<td>250</td>
<td>100</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.00</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>28.11</td>
<td>N</td>
</tr>
<tr>
<td>600</td>
<td>200</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.50</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>169.08</td>
<td>N</td>
</tr>
<tr>
<td>800</td>
<td>200</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12.79</td>
<td>N</td>
</tr>
</tbody>
</table>

### Table 7.2: RoadCon

<table>
<thead>
<tr>
<th></th>
<th>CL+GR</th>
<th>CHUFFED</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gr opt comp time</td>
<td>flat opt comp time</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>45</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>25</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 7.3: CompCon

<table>
<thead>
<tr>
<th></th>
<th>CL+GR</th>
<th>CHUFFED</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gr opt comp time</td>
<td>flat opt comp time</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.01</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>66.64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
</tr>
<tr>
<td>20</td>
<td>25</td>
<td>89.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.06</td>
</tr>
<tr>
<td>90</td>
<td>90</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.40</td>
</tr>
<tr>
<td>250</td>
<td>230</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13.74</td>
</tr>
</tbody>
</table>

### Table 7.4: UtilPol comparison, and scaling behaviour on the smallest instance

<table>
<thead>
<tr>
<th></th>
<th>gr kB CL+GR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>748 0.36</td>
</tr>
<tr>
<td>2</td>
<td>2816 1.25</td>
</tr>
<tr>
<td>3</td>
<td>8252 9.39</td>
</tr>
<tr>
<td>4</td>
<td>18741 37.83</td>
</tr>
<tr>
<td>5</td>
<td>36288 114.22</td>
</tr>
<tr>
<td>6</td>
<td>67570 260.39</td>
</tr>
<tr>
<td>7</td>
<td>113475 498.09</td>
</tr>
<tr>
<td>8</td>
<td>170287 600</td>
</tr>
<tr>
<td>9</td>
<td>243608 600</td>
</tr>
<tr>
<td>10</td>
<td>336386 600</td>
</tr>
</tbody>
</table>
Chapter 8
Language of BFASP

In this chapter, we look at representing and simplifying large BFASPs. We first introduce non-ground BFASPs which allow us to group similar variables in arrays and similar rules (constraints) into a single rule expressed in terms of array variables. Next, we look at the problem of flattening where we reduce all rules and constraints in a BFASP to another BFASP in which all constraints and rules are composed of a given set of primitive expressions. The other two problems are related to grounding BFASPs efficiently in order to get BFASPs that are equivalent to the ones that we can get by enumerating the non-ground generator expressions naively, but have potentially smaller set of variables and rules (constraints). We borrow two grounding techniques from logic programming, namely bottom-up grounding or semi-naive evaluation, and magic set transformation, and extend them to BFASP.

8.1 Non-Ground BFASPs

A non-ground BFASP is a BFASP where sets of variables are grouped together in variable arrays, and sets of ground rules are represented by non-ground rules via universal quantification over index variables. For example, if we have arrays of variables $a, b, c$, then we can represent the ground rules: $(a[1] \geq b[1] + c[1], a[1]), (a[2] \geq b[2] + c[2], a[2]), (a[3] \geq b[3] + c[3], a[3])$ by $\forall i \in [1,3] : (a[i] \geq b[i] + c[i], a[i])$. Variables can be grouped together in arrays of any dimension and non-ground BFASP rules have the following form: $\forall \vec{i} \in \vec{D}$ where $\text{con}(\vec{i}) : (y[l_0(\vec{i})] \geq f(x_1[l_1(\vec{i})], \ldots, x_n[l_n(\vec{i})]), y[l_0(\vec{i})])$, where $\vec{i}$ is a set of index variables $i_1, \ldots, i_m$, $\vec{D}$ is a set of domains $D_1, \ldots, D_m$, $\text{con}$ is a constraint over the index variables which constrains these variables, $l_0, \ldots, l_n$ are functions over the index variables which return a tuple of array indices, $y, x_1, \ldots, x_n$ are arrays of variables and $f$ is a function over the $x_i$ variables. Let $\text{gen}(r) \equiv \vec{i} \in \vec{D} \land \text{con}(\vec{i})$ denote the generator.
constraint for a non-ground rule $r$. Note that we require the generator constraint in each rule to constrain the index variables so that $f$ is always defined.

**Example 8.1.** Suppose we have a 2 dimensional array of variables $a$ and 1 dimensional arrays of variables $b$ and $c$. In the non-ground rule: \( \forall i, j \in [1, 10] \) where \( i < j \) : \( a[i,j] \geq b[i+1] + c[i \ast j], a[i,j] \), $i$ and $j$ are the index variables, \( i, j \in [1, 10] \land i < j \) is the generator constraint, \( l_0(i,j) = (i,j), l_1(i,j) = (i + 1), l_2(i,j) = (i \ast j) \), and $f(x_1, x_2) = x_1 + x_2$.

Note that we require the generator constraint in each rule to constrain the index variables so that $f$ is always defined. For example, if we have variables $a[1], \ldots, a[10], b[1], \ldots, b[10]$ and $c[1], \ldots, c[10]$, then the rule $\forall i \in [1, 10] (a[i] \geq b[i + 1] + c[i + 2], a[i])$ is not valid since $c[i + 2]$ refers to a variable outside the array when $i = 10$. On the other hand, $\forall i \in [1, 9] : (a[i] \geq b[i + 1] + c[i + 2], a[i])$ is valid since all values of $i$ refer to variables within the array. We can relax this restriction but it requires us to carefully treat partial function applications (see e.g. related work by Frisch and Stuckey (2009)). An exception to this occurs when the function in the rule body supports undefined arguments. For example, we can define $\text{max}(x_1, \ldots, x_n)$ to be $-\infty$ if all the arguments are undefined and the max of the ones which are defined when at least one is defined. Then the rule $\forall i \in [1, 10] (a[i] \geq \text{max}(b[i + 1], c[i + 2]), a[i])$ is valid since when $b[i + 1]$ or $c[i + 2]$ refer to a variable that does not exist, we simply consider that as an undefined argument and the function is still defined so the rule is valid.

Variable arrays can contain either founded variables, standard variables, or parameters (which can simply be considered fixed standard variables), although all variables in a variable array must be of the same type. Note that the array names in our notation correspond to predicate names in standard ASP syntax, and our index variables correspond to ASP local or first-order variables. Also, in standard ASP syntax, the generator constraint is often put in the rule body instead of being made explicit. We make them explicit so that we do not have to mix Boolean conditions with arithmetic functions in the rule body. For example, an ASP rule: $\text{rch}(X) \leftarrow \text{rch}(Y), e(X,Y), \text{node}(X), \text{node}(Y)$ would be written in our syntax as:

\[
\forall x \in \text{Node}, y \in \text{Node} \text{ where } e[x, y] : (\text{rch}[x] \leftarrow \text{rch}[y], \text{rch}[x])
\]
8.2 Flattening

Let us begin by observing that an arithmetic function can be written as an abstract syntax tree as illustrated in the following example.

**Example 8.2.** The function \( f(x_1, \ldots, x_5) = x_1 + \min(x_2, x_3 - x_4) - (x_5)^2 \) can be described by the tree given below.

![Example Tree]

A ground BFASP may contain constraints and rules whose expressions are not flat, i.e., they are expression trees with height greater than one. Such expressions are not supported by constraint solvers and we need to flatten these expressions to primitive

where \( e(X, Y) \) and \( e[x, y] \) are fixed relations in the data that is given in the ASP and BFASP program respectively.

Given a non-ground rule \( r \), let \( \text{grnd}(r) \) be the set of ground rules obtained by substituting all possible values of the index variables that satisfy \( \text{gen}(r) \) into the quantified expression. Similarly given a non-ground BFASP \( P \), let \( \text{grnd}(P) \) be the grounded BFASP that contains the grounding of all its rules and constraints. The *predicate dependency graph*, validity and stratification are defined similarly for array variables and non-ground rules as the *local* dependency graph, local validity and local stratification respectively are defined for ground variables and ground rules (see 6.3). All our subsequent discussion is restricted to valid BFASPs. Note that similarly to how local stratification is a tighter condition for logic programs as compared to local validity, local validity is also a tighter condition for BFASPs.
forms. We omit consideration of flattening constraints since this is the same as in standard CP (Stuckey and Tack, 2013). Consider the expression tree in Example 8.2, if it were a constraint, we would introduce variables \( i_1, \ldots, i_5 \) to decompose the given function into the following set of equalities: \( f = x_1 + i_1 + i_2, i_1 = \min(x_2, i_3), i_3 = x_3 + i_4, i_4 = -x_4, i_2 = -i_5, i_5 = x_5 \times x_5 \).

It can be shown that the standard CP flattening approach in which a subexpression is replaced with a standard variable and a constraint is added that equates the introduced variable with the subexpression, does not preserve stable model semantics. If a standard variable is introduced in order to represent a subexpression containing founded variables, the stable solutions of the program may change.

**Example 8.3.** Consider a BFASP with rules: \((x_1 \geq \max(x_2, x_3) - 2, x_1), (x_2 \geq x_1 + 1, x_2), (x_3 \geq x_1 + 2, x_3), (x_1 \geq 3, x_1)\) where \(x_1, x_2, x_3\) are all founded variables. The only stable solution of this program is \(x_1 = 3, x_2 = 4, x_3 = 5\). Suppose we introduced a standard variable \( i_1 \) to represent the subexpression \(\max(x_2, x_3)\), so that the first rule in the program is replaced by: \((x_1 \geq i_1 - 2, x_1)\) and \(i_1 = \max(x_2, x_3)\). Now, due to the introduction of the standard variable \(i_1\), the new program has many new spurious stable solutions such as \(i_1 = 6, x_1 = 4, x_2 = 5, x_3 = 6\) or \(i_1 = 7, x_1 = 5, x_2 = 6, x_3 = 7\) etc.

To preserve the stable model semantics, it is necessary to use introduced *founded* variables to represent subexpressions containing founded variables. Note that it is still correct to replace a founded variable with a standard variable if that variable is not in the same SCC in the positive dependency graph as the head of the rule. Therefore, for all subsequent discussion related to flattening in this Section, we treat founded variables from different SCCs as standard variables. We now describe the central result used in our flattening algorithm.

**Theorem 8.1.** Let \( P \) be a BFASP containing a rule \( r = (y \geq f_1(x_1, \ldots, x_k, f_2(x_{k+1}, \ldots, x_n)), y) \) where \( f_1 \) is increasing in the argument where \( f_2 \) appears, and where if a variable occurs among both \(x_1, \ldots, x_k\) and \(x_{k+1}, \ldots, x_n\), then \( f_1 \) and \( f_2 \) have the same monotonicity w.r.t. it. Let \( P' \) be \( P \) with \( r \) replaced by the two rules: \( r_1 = (y \geq f_1(x_1, \ldots, x_k, y'), y) \) and \( r_2 = (y' \geq f_2(x_{k+1}, \ldots, x_n), y') \) where \( y' \) is an introduced founded variable. Then the stable solutions of \( P' \) restricted to the variables of \( P \) are equivalent to the stable solutions of \( P \).
8.2 Flattening

Proof. For a rule \( s = (c, \text{head}) \), let \( \text{con}(s) = c \). By construction, \( \text{con}(r) \Leftrightarrow \exists y'(\text{con}(r_1) \land \text{con}(r_2)) \) and all the other constraints in \( P \) and \( P' \) are identical. Also, given any assignment \( \theta' \) of \( P' \), since \( f_1 \) is increasing in the argument where \( f_2 \) appears, \( y' \) will be left as a variable in \( r_1^{\theta} \). Consider an assignment \( \theta' \) over \( \text{vars}(P') \), and let \( \theta = \theta'|_{\text{vars}(P)} \). Recall that the reduct of a program with respect to an assignment replaces all the standard variables and founded variables that are not decreasing in any rule’s constraint with its value in that assignment. Since \( f_1 \) and \( f_2 \) have the same monotonicity w.r.t. any variable common in \( \{x_1, \ldots, x_k\} \) and \( \{x_{k+1}, \ldots, x_n\} \), it will either be replaced by its assignment value in both \( f_1 \) and \( f_2 \) or not be replaced at all. Therefore, the relation \( r^\theta \Leftrightarrow \exists y'(r_1^{\theta'} \land r_2^{\theta'}) \) is also valid. Furthermore, all other constraints in \( P^\theta \) and \( P'^{\theta'} \) are identical.

Suppose \( \theta \) is a stable solution of \( P \). Let \( \theta' \) be the extension of \( \theta \) to variable \( y' \) s.t. \( \theta'(y') = f_2(\theta(x_k), \ldots, \theta(x_n)) \). Clearly, this choice of \( \theta'(y') \) allows \( \theta' \) to satisfy all the constraints of \( P' \) and allows \( \theta'|_{\text{vars}(P')} \) to satisfy all the constraints of \( P'^{\theta'} \). To prove that \( \theta' \) is a stable solution of \( P' \), we just need to show that there is no smaller solution of \( P'^{\theta'} \) than \( \theta'|_{\text{vars}(P')} \). Since \( r^\theta \Leftrightarrow \exists y'(r_1^{\theta'} \land r_2^{\theta'}) \) and all other constraints in \( P^\theta \) and \( P'^{\theta'} \) are identical, \( P'^{\theta'} \) must force the same lower bounds on the variables in \( \text{vars}(P^\theta) \) as \( P^\theta \) does. Hence, none of those values can go any lower. Also, \( r_2^{\theta'} \) forces \( y' \geq f_2(\theta(x_k), \ldots, \theta(x_n)) \), and so \( f_2(\theta(x_k), \ldots, \theta(x_n)) \) is the lowest possible value for \( y' \). Hence \( \theta'|_{\text{vars}(P')} \) is the minimal solution of \( P'^{\theta'} \) and \( \theta' \) is a stable solution of \( P' \).

Suppose \( \theta' \) is a stable solution of \( P' \). Let \( \theta = \theta'|_{\text{vars}(P)} \). Since \( \text{con}(r) \Leftrightarrow \exists y'(\text{con}(r_1) \land \text{con}(r_2)) \) and all the other constraints in \( P \) and \( P' \) are identical, \( \theta \) satisfies all the constraints in \( P \). Since \( r^\theta \Leftrightarrow \exists y'(r_1^{\theta'} \land r_2^{\theta'}) \) and all other constraints in \( P^\theta \) and \( P'^{\theta'} \) are identical, \( \theta|_{\text{vars}(P')} \) satisfies all the constraints in \( P^\theta \). To prove that \( \theta \) is a stable solution of \( P \), we just need to show that there is no smaller solution of \( P^\theta \) than \( \theta|_{\text{vars}(P')} \). Since \( r^\theta \Leftrightarrow \exists y'(r_1^{\theta'} \land r_2^{\theta'}) \) and all other constraints in \( P^\theta \) and \( P'^{\theta'} \) are identical, \( P^\theta \) must force the same lower bounds on the variables in \( \text{vars}(P^\theta) \) as \( P'^{\theta'} \) does. Hence, none of those values can go any lower, \( \theta|_{\text{vars}(P')} \) is the minimal solution of \( P^\theta \) and \( \theta \) is a stable solution of \( P \).

Theorem 8.1 tells us that we can preserve the stable model semantics by introducing a founded variable to represent subexpressions containing founded variables. As a corollary, if \( f_1 \) is decreasing in the argument where \( f_2 \) appears, we can replace \( f_2 \) by a
founded variable $-y'$ and add the rule $(y' \geq -f_2(x_k, \ldots, x_n), y')$ instead. Not all valid rule forms are supported by Theorem 8.1, because we require that multiple occurrences of the same variable in the expression must have the same monotonicity w.r.t. the root expression.\footnote{For replacing body arguments, these results can easily be generalized even when the rules do not adhere to the restriction in Section 6.4. Now we discuss two other corollaries when the head argument is not primitive, that is when the rule has the form: $(c(f(y), \bar{x}), y)$. We reiterate the requirement that every occurrence of a founded variable has the same monotonicity w.r.t. to the root and $c$ is increasing in the head $y$. The first corollary is for the case when $c_{f(y)}$. In this case, we can replace the rule with these two rules: $(c(r, \bar{x}), r)$ and $(f(y) \geq r, y)$. For example, consider the rule where we have all positive founded integers, and the rule: $(bc \geq 2a - 3d, b)$; we can replace this with these two rules: $(r \geq 2a - 3d, r)$ and $(bc \geq r, b)$ (we could rewrite the second rule as $(b \geq \frac{1}{r}, b)$). The second case is when $c_{f(y)}$. Here, we can replace the rule with these two rules: $(c(-r, \bar{x}), r)$ and $(-r \geq f(y), y)$. For example, consider the rule: $(a \geq \min(\frac{b}{y}, z), y)$ when $b$ and $y$ are positive founded integers. The constraint is decreasing in the RHS, but the RHS is decreasing again in $y$, therefore, the constraint (root) is increasing in $y$. Using the transformation just described, we could use the following two rules in its place: $(a \geq \min(-r, z), r)$ and $(-r \geq \frac{b}{y}, y)$ (the second one could be written as $(y \geq \frac{b}{-r}, y)$.)}
8.2 Flattening

\[
\text{flat}(P)
\]

\[
P_{\text{flat}} := \emptyset
\]

\[
R := \text{rules}(P)
\]

\[
T := \text{constraints}(P)
\]

\[
\text{for } (r \in R)
\]

\[
R := R \setminus \{r\}
\]

\[
\text{flatRule}(r, R, T)
\]

\[
r := \text{simplify}(r)
\]

\[
P_{\text{flat}} \cup \{r\}
\]

\[
\text{for } (c \in T) P_{\text{flat}} \cup = \text{cp}_{\text{flat}}(c)
\]

\[
\text{return } P_{\text{flat}}
\]

\[
\text{flatRule}(r = (y \geq f(e_1, \ldots, e_n), y), R, T)
\]

\[
\text{for } (\text{each non-terminal } e_i)
\]

\[
\text{if } (e_i \text{ does not contain founded vars})
\]

\[
\text{replace } e_i \text{ with standard var } y' \text{ in } r
\]

\[
T \cup = \{y' = e_i\}
\]

\[
\text{elif } (f \text{ is increasing in } e_i)
\]

\[
\text{replace } e_i \text{ with founded var } y' \text{ in } r
\]

\[
R \cup = \{(y' \geq e_i, y')\}
\]

\[
\text{elif } (f \text{ is decreasing in } e_i)
\]

\[
\text{replace } e_i \text{ with founded var } -y' \text{ in } r
\]

\[
R \cup = \{(y' \geq -e_i, y')\}
\]

Example 8.4. Consider the rule: \((y \geq x_1 + \min(x_2, x_3 - x_4) - (x_5)^2, y)\) where \(x_1, x_2, x_5\) are founded and \(x_3, x_4\) are standard variables. Using our flattening algorithm, we can break the rule into: \((y \geq x_1 + i_1 + i_2, y), (i_1 \geq \min(x_2, x_3 - x_4), i_1), (i_2 \geq -(x_5)^2, i_2)\) where \(i_1, i_2\) are founded variables. The rule \((i_1 \geq \min(x_2, x_3 - x_4), i_1)\) is further flattened to \((i_1 \geq \min(x_2, i_3), i_1)\) and a constraint \(i_3 = x_3 - x_4\) where \(i_3\) is a standard variable. □

Once we have flattened the entire program, we can calculate the initial domains for the introduced variables as well as the \(ujb\) values for the introduced founded variables. This step is not strictly necessary for correctness since we can always set the \(ujb\) values of founded variables to \(-\infty\) and set the initial domains of introduced variables to \((-\infty, \infty)\). However, getting tighter initial domains may allow us to get a smaller grounding and/or improve solving efficiency, when used with the grounding algorithms described in the
subsequent sections. We perform these calculations in the reverse order in which the variables were introduced. Suppose \( y' \) is an introduced founded variable. There will be exactly one rule \( (y' \geq f(x_1, \ldots, x_m), y') \) with \( y' \) as head in the flattened program, which means that the stable model semantics will actually force \( y' \) to be equal to \( f(x_1, \ldots, x_m) \).

Finding the initial domains for the introduced variables then simply requires propagating the constraints and is standard in CP flattening so we do not discuss this further. We also have to calculate \( ujb(y') \). To do this, we take the expression \( f(x_1, \ldots, x_n) \). For each founded argument \( x_i \) where \( f \) is increasing in \( x_i \), we substitute \( ujb(x_i) \) for \( x_i \). We then perform propagation on the modified \( f(x_1, \ldots, x_n) \) to find a lower bound for this expression. That lower bound value is then used as \( ujb(y') \).

**Example 8.5.** Assuming the initial domains of all variables were \([0, 10]\), and \( ujb(x_1) = 1\), \( ujb(x_2) = 2\) and \( ujb(x_3) = 5\), then the domains of the introduced variables of Example 8.3 are: \( i_3 \in [-10, 10]\), \( i_2 \in [-100, 0]\), \( i_1 \in [-10, 10]\) and the ujb values are: \( ujb(i_2) = -25\) and \( ujb(i_1) = -10\).

The algorithm can be extended to non-ground rules by defining the index set of the introduced variables to be equal to the domain of index variables as given in the generator of the rule in which they replace an expression. Moreover, the generator expression of an intermediate rule stays the same as that of the original rule from which it is derived. Establishing initial domains and bounds on introduced variables is the same as it is for the ground counterpart.

**Example 8.6.** Consider a nonground BFASP \( P \) with arrays of founded variables, \( a, b, c, d \), all with index sets \([1, 100]\) and initial bounds of \(-\infty\), and a rule \( r = (c, a[i]) \) where \( c \) is:

\[
\forall i \in [2,30] \text{ where } i \mod 2 = 0 : \\
\begin{align*}
a[i] &\geq b[i] - 1 + \max(c[2i], d[i+1])
\end{align*}
\]
After flattening, we get the following rules and an intermediate variable $y_1$.

\[
\text{gen}(r): a[i] \geq b[i-1] + y_1[i]
\]

\[
\text{gen}(r): y_1[i] \geq \max(c[2i], d[i+1])
\]

The index set for $y_1$ is $[2, 30]$ and its initial bound is equal to $-\infty$. □

### 8.3 Grounding

ASP grounders keep track of variables that have been created and instantiate further rules based on that. For example, if the variables $b$ and $c$ have been created, then the rule $a \leftarrow b \land c$ justifies a bound on $a$ and therefore, must be included in the final program.

The justification of all positive literals in a rule potentially justify its head. However, for a rule, if any one positive variable in its body does not have any rule supporting it, then that rule can safely be ignored until a justification for that variable has been found. In case a justification is never found for that variable, then the rule is useless, i.e., excluding the rule from the program does not change its stable solutions.

We propose a simple grounding algorithm for non-ground BFASPs which can be implemented by simply maintaining a set of ground rules and variables as done in ASP grounders, but which may generate useless rules in addition to all the useful ones. The idea is that for each variable $v$, we only keep track of whether $v$ can potentially be justified above its $ujb$ value, rather than keeping track of whether it can be justified above each value in its domain. If it can be justified above its $ujb$, then when $v$ appears in the body of a rule, we assume that $v$ can be justified to any possible bound for the purpose of calculating what bound can be justified on the head. This clearly over-estimates the bounds which can be justified on the variables, and thus the algorithm generates all the useful rules and possibly some useless ones.

We refer to a variable $x$ as being created, written $cr(x)$, if it can go above its $ujb$ value. More formally, $cr(x)$ is a founded Boolean with a rule: $cr(x) \leftarrow x > ujb(x)$. While that is how we define $cr(x)$, we do not explicitly have a variable $cr(x)$ or the above rule in our
Table 8.1: Grounding conditions for rule \( r = (c, y) \)

<table>
<thead>
<tr>
<th>( c )</th>
<th>( \phi_r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y \geq \text{sum}(x_1, \ldots, x_n) )</td>
<td>((\sum_i ujb(x_i) &gt; ujb(y)) \lor ((\land_i ujb(x_i) &gt; -\infty \lor cr(x_i)) \land (\lor_i cr(x_i))))</td>
</tr>
<tr>
<td>( y \geq \text{max}(x_1, \ldots, x_n) )</td>
<td>( \lor_i (ujb(x_i) &gt; ujb(y) \lor cr(x_i)) \land_i (ujb(x_i) &gt; ujb(y) \lor cr(x_i)) )</td>
</tr>
<tr>
<td>( y \geq \text{min}(x_1, \ldots, x_n) )</td>
<td>( \land_i (ujb(x_i) &gt; ujb(y) \lor cr(x_i)) \lor_i (ujb(x_i) &gt; ujb(y) \lor cr(x_i)) )</td>
</tr>
<tr>
<td>( y \geq \text{product}(x_1, \ldots, x_n) ) where ( \land_i x_i &gt; 0 )</td>
<td>( \land_i (ujb(x_i) &gt; ujb(y)) \lor (\lor_i cr(x_i))) )</td>
</tr>
<tr>
<td>( y \geq x \leftarrow r )</td>
<td>( cr(r) \land (ujb(x) &gt; ujb(y) \lor cr(x)) )</td>
</tr>
<tr>
<td>( y \leftarrow x \geq 0 )</td>
<td>( ujb(x) \geq 0 \lor cr(x) )</td>
</tr>
<tr>
<td>( y \leftarrow \land_i x_i )</td>
<td>( \land_i cr(x_i) )</td>
</tr>
<tr>
<td>( y \leftarrow \lor_i x_i )</td>
<td>( \lor_i cr(x_i) )</td>
</tr>
<tr>
<td>( y \geq -x )</td>
<td>( -ub(x) &gt; ujb(y) )</td>
</tr>
<tr>
<td>( y \leftarrow \neg x )</td>
<td>( \text{true} )</td>
</tr>
<tr>
<td>( y \geq 1/x ) where ( x &gt; 0 )</td>
<td>( 1/ - ub(x) &gt; ujb(y) )</td>
</tr>
</tbody>
</table>

implementation. Instead, we implement it by maintaining a set \( Q \) of variables that have been created. Initially, \( Q \) is empty. We recursively look at each non-ground rule to see if the newly created variables make it possible for more head variables to be justified above their \( ujb \) values. If so, we create those variables and add them to \( Q \). In order to do this, we need to find necessary conditions under which the head variable can be justified above its \( ujb \). In order to simplify the presentation, we are going to define \( ujb \) for constants, standard variables and parameters as well. For a constant \( x \), we define \( ujb(x) \) to be the value of \( x \). For parameters and standard variables \( x \), we define \( ujb(x) = ub(x) \).\(^{25}\) Note that for soundness, the \( ujb \) values of founded variables only have to be correct (e.g. \( -\infty \) for all variables) although tighter \( ujb \) values can improve the efficiency of our algorithm. Table 8.1 gives a non-exhaustive list of necessary conditions for the head variable to be justified above its \( ujb \) value for different rule forms.

Let us now make a few observations about the conditions given in Table 8.1. A key

\(^{25}\)Upper and lower bounds for a parametric array can be established by simply parsing the array.
point is that for many rule forms \( \phi_r \) can evaluate to true, even without any variable in the body getting created. For example, for max, even if one variable has a \( \text{ujb} \) value that is greater than the \( \text{ujb} \) of the head, the rule needs to be grounded completely. All such rules that evaluate to true give us a starting point for initializing \( Q \) in our implementation. For an ASP program, this means initializing \( Q \) with facts. The linear case (sum) deserves some explanation. It is made up of two disjuncts, the first of which is an evaluation of the initial condition, i.e., whether the sum of \( \text{ujb} \) values of all variables is greater than the \( \text{ujb} \) of the head. If this condition is true, then the rule needs to be grounded unconditionally. If this is false, then the second disjunct becomes important. The second disjunct itself is a conjunction of two more conditions. The first one says that all variables must be greater than \(-\infty\) in order for the rule to justify a finite value on the head. In the case where all variables already have a finite \( \text{ujb} \), the second conjunct says that at least one of them must be created for the rule to be grounded (given the initial condition failed). Note that this condition becomes redundant if at least one of the variables has a \( \text{ujb} \) of \(-\infty\). Finally, observe that after plugging all values of \( \text{ujb} \), all conditions given in the table simplify to one of the following four forms: true, false, \( \lor \bigwedge_i \text{cr}(x_i) \) or \( \land \text{cr}(x_i) \). Note that the grounding conditions are significantly more sophisticated than the simple conjunctive condition for normal rules. More specifically, after simplification, we can get a disjunctive condition which has no analog in ASP.

**Example 8.7.** Consider a BFASP with the following two non-ground rules: \( \forall i \in [1, 10] : a[i] \geq b[i] + x[i] \) and \( \forall i \in [1, 10] : x[i] \geq \min(c[i], d[i]) \). Say \( \text{ujb}(a) = 5, \text{ujb}(b) = 2, \text{ujb}(c) = 7, \text{ujb}(d) = 1 \) and \( \text{ujb}(x) = 1 \). For the first rule, the initial condition evaluates to false. Moreover, since both \( b \) and \( x \) have \( \text{ujb} \) greater than \(-\infty\), we get \( \text{cr}(b[i]) \lor \text{cr}(x[i]) \). For the second rule, since \( \text{ujb}(c[i]) > \text{ujb}(x[i]) \) and \( \text{ujb}(d[i]) \) is not greater than \( \text{ujb}(x[i]) \), we get the condition: \( \text{cr}(d[i]) \).

We are now ready to present the main bottom-up grounding algorithm. Logically, our grounding algorithm starts with \( \text{ujb}(x) \) for all \( x \), adds \( (x \geq \text{ujb}(x), x) \) to the program and then finds all the ground rules that are not made redundant by these rules. \text{createCPs} is a preprocessing step that creates constraint programs for rules in a BFASP \( P \) whose conditions are either conjunctions or disjunctions. For a rule with a conjunctive condition,
createCPs(P)
  for (r ∈ rules(P) : φr = \bigwedge_{i=1}^{n} cr(x_i[\overline{l}_i]))
  
  cp[r] := true % new constraint program
  cp[r] := cp[r] \land gen(r)
  for (i ∈ 1...n)
    set[r,i] := \emptyset
    cp[r] := cp[r] \land \overline{l}_i \in \ll set[r,i] \gg

  for (r ∈ rules(P) : φr = \bigvee_{i=1}^{n} cr(x_i[\overline{l}_i]))
    for (i ∈ 1...n)
      cp[r,i] := true % new constraint program
      cp[r,i] := cp[r,i] \land gen(r)
      set[r,i] := \emptyset
      cp[r,i] := cp[r,i] \land \overline{l}_i \in \ll set[r,i] \gg

Figure 8.1: Creating constraint programs for grounding rules

it only creates one program, while for one with a disjunctive condition, it creates one constraint program for each variable in the condition. Each program is initialized with the \textit{gen(r)} which defines the variables and some initial constraints given in the where clause in the generator of non-ground rule. Furthermore, for each array literal in \( \phi_r \), a constraint is posted on its literal (which is a function of index variables in the rule), to be in the domain given by the \textit{current} value of the \textit{set} variable (the reason for the Quine quotes) which is initially set to empty. \textit{ground} is called after preprocessing. \( Q \) and \( R \) are sets of ground variables and rules respectively. \textit{groundAll} is a function that grounds a non-ground rule or constraint completely, and returns the set of all rules and constraints respectively. Initially, we ground all constraints in \( P \) and rules for which \( \phi_r \) evaluates to true. \( R' \) is a temporary variable that represents the set of new ground rules from the last iteration. In each iteration, we only look for non-ground rules that have some variable in their conditions that is created in the previous iteration. \textit{heads} takes a set of ground rules as its input and returns their heads. In each iteration, through \( Q \), we manipulate the \textit{set} constraint to get new rule instantiations. For each variable in the clause, we make \textit{set} equal to the new index values created for that variable. For both the conjunctive and the disjunctive case, this optimization only tries out new values of recently created
8.4 Magic Set Transformation

In this section, we describe a useful grounding technique developed for logic programs, called magic set rewriting or magic set transformation (Bancilhon et al., 1985, Beeri and Ramakrishnan, 1991), for BFASPs. In the first subsection, we briefly describe the background of magic sets in the context of logic programming, and illustrate why the technique is useful in logic programming, and in the next subsection we discuss magic set transformation for BFASPs.

variables to instantiate new rules. search takes a constraint program as its input, finds all its solutions, instantiates the non-ground rule for each solution, and returns the set of these ground rules. After creating new rules due to the new values in set, we make it equal to all values of the variable in Q. The fixed point calculation stops when no new rules are created. Finally, for every founded variable y, we add \((y \geq ujb(y), y)\) as a rule so that if the ujb relied on some rules that were ignored during grounding, then this ensures that ujb(y) is always justified.

Figure 8.2: Pseudo-code for bottom-up grounding

ground(P)
C := \{groundAll(c) : c ∈ constraints(P)\}
R′ := \{groundAll(r) : r ∈ rules(P) : \(φ_r = true\)\}
while \(R′ \neq ∅\)

H := heads(R′)
Q := H
R′ := ∅
for \(r ∈ rules(P) : H \cap vars(φ_r) \neq ∅\)

if \(φ_r = \bigwedge_{i=1}^n cr(x_i[l_i]) \lor φ_r = \bigvee_{i=1}^n cr(x_i[l_i])\)

for \((i ∈ 1...n)\)

dom := \{m | x[m] ∈ Q\}
set[r, i] := dom \setminus \{r[i]\}
if \(φ_r\) is conj \(R′ := search(cp[r]) \setminus R\)
if \(φ_r\) is disj \(R′ := search(cp[r,i]) \setminus R\)
R := R \\
set[r, i] := dom
for \((y ∈ \text{vars}(R) \cap V_F)\) \(R \cup=(y \geq ujb(y), y)\)
8.4.1 Background

Consider stratified logic programs with no standard variables (traditional logic programs), a property of such a program is that it has a unique stable model (Gelfond and Lifschitz, 1988b). Note that stratified logic programs do not have any constraints (recall the integrity constraint representation described in Section 2.4.1 and note that it introduces unstratified negation via the auxiliary variable $f$). Say that we wish to query whether a certain variable $a$ is true in that unique stable model. Also, imagine that there are million other rules in the program that are not reachable from $a$ in the dependency graph of the program. That is, if we start from rules where $a$ is the head, then go over the variables in bodies of those rules, and then look at rules for those variables, and so on, we can never reach any of those million other rules. A traditional top-down logic programming solver takes a similar approach to answering the query $a$. We do not go into details of the algorithm that is used, called Selective Linear Definite-clause with Negation as Failure (SLDNF) resolution (Cavedon and Lloyd, 1989), since they are not relevant for this section. We are only interested in the fact that top-down computation towards answering a query can be very useful in cases when there are large parts of logic program that have nothing to do with the truth value of the query. Let us look at an example to illustrate some of the important points about top-down based approach.

Example 8.8. Consider the stratified logic program with following rules (we are dropping the head argument from rules and all variables are founded):

\[
\begin{align*}
  a & \leftarrow b \land \neg c \\
  b & \\
  c & \leftarrow d \\
  d & \leftarrow \neg e \\
  e & \\
  j & \leftarrow i \land \neg k \land \neg u \\
  k & \leftarrow t \land s \land \neg b \\
  s & \leftarrow r \land \neg z \\
  z & \leftarrow e \land a \\
  r & \leftarrow \neg y
\end{align*}
\]

The program has the unique stable assignment: \{a, b, e, \neg c, \neg d, z, r, \neg s, \neg u, \neg k, \neg j, \neg i\}. Say we are interested in finding out if $a$ is true in this stable assignment, a top-down solver would work along these lines: since $a$ only has one rule, $b$ has to be true and $c$ has to be false to derive $a$. $b$ is true, so let us focus on proving that $c$ cannot be true; in order for $c$ to be false, $d$ must also be false because of the rule $c \leftarrow d$ and in order for $d$ to be false, $e$ must be true (rule $d \leftarrow \neg e$), and
since e is given as a fact, d is false, which means that c is false, which means that a is true, and we are done. The solver has derived this part of the unique stable assignment: \{a, b, e, ¬c, ¬d, \} without even having to consider the entire second row of rules! We do not care about that row and variables that are only in that row, since as long as the program is stratified, we can be sure that there is a way to extend our partial stable assignment to a complete one.

There are a couple of points worth noting. First, say we change the program and delete the only rule for b, then in the derivation above, we do not even have to look for ways to prove ¬c. The fact that b cannot be derived is enough to set a to false, since ¬b is sufficient to prove that the body of the only rule of a is false. The relevant assignment would be \{¬a, ¬b\} and the answer query a would be false (or no). Second, note the direction of our interest goes from head to body. In this example, the variables that we care about are present in the rule bodies in the second row, but we are not interested in those rules.

Magic sets technique is an attempt to simulate top-down solving using bottom-up grounding. The core idea is that we transform the program in a way such that when we perform bottom-up grounding, the irrelevant rules become useless. More particularly, for every variable a in the original program, we create a magic variable \(m_a\) that represents whether we care about a. Additionally, there are magic rules defined for every normal rule that capture relations between these magic variables. For every query variable q, we set \(m_q\) to true. For a normal rule \(a ← b_1 ∧ \ldots b_j ∧ ¬c_1 ∧ \ldots ∧ c_k\), we create the following magic rules:

\[
\begin{align*}
  m_{b_1} &← m_a \\
  m_{b_2} &← m_a ∧ b_1 \\
  &\ldots \\
  m_{b_j} &← m_a ∧ b_1 ∧ \ldots ∧ b_{j-1} \\
  m_{c_1} &← m_a ∧ b_1 ∧ \ldots ∧ b_j \\
  &\ldots \\
  m_{c_k} &← m_a ∧ b_1 ∧ \ldots ∧ b_j
\end{align*}
\]

Finally, we modify the original rule so that it is only included in the program if we are interested in the head variable:
After performing bottom-up grounding on the transformed program, the final step is to get rid of magic variables and magic rules.

In Example 8.8, we looked at a ground program. But the magic set technique is much more useful for non-ground programs. We demonstrate that in the next example. A point worth mentioning is that in the original magic set technique, the query can contain free first-order variables, but we do not allow that and only consider ground queries. Therefore, we do not need to propagate binding information as in the original magic set technique. The original magic set technique has three stages: adorn, generate and modify. For the reason described above, we do not describe adornment, and we have already described the generation of magic rules and modification of the original rules. The following example demonstrates the use of magic set transformation on non-ground programs.

**Example 8.9.** Consider the following non-ground logic program:

\[
\forall i \in 1 \ldots 10000 : a[i] \leftarrow b[i] \land b[i + 1] \land \neg c[i]
\]

\[
\forall i \in 1 \ldots 10000 : b[i]
\]

\[
\forall i \in 1 \ldots 10000 : c[i] \leftarrow \neg e[i]
\]

The magic rules, as described above, are:

\[
\forall i \in 1 \ldots 10000 : m._b[i] \leftarrow m._a[i]
\]

\[
\forall i \in 1 \ldots 10000 : m._b[i + 1] \leftarrow m._a[i] \land b[i]
\]

\[
\forall i \in 1 \ldots 10000 : m._c[i] \leftarrow m._a[i] \land b[i] \land b[i + 1]
\]

\[
\forall i \in 1 \ldots 10000 : m._e[i] \leftarrow m._c[i]
\]

After modifying our original program as described, the final non-ground program is given
below. We have given rules names, so that we can refer to them.

\begin{align*}
R_1 & : \forall i \in 1 \ldots 10000 : a[i] \leftarrow m_a[i] \land b[i] \land b[i+1] \land \neg c[i] \\
R_2 & : \forall i \in 1 \ldots 10000 : b[i] \leftarrow m_b[i] \\
R_3 & : \forall i \in 1 \ldots 10000 : c[i] \leftarrow m_c[i] \land \neg e[i] \\
R_4 & : \forall i \in 1 \ldots 10000 : m_b[i] \leftarrow m_a[i] \\
R_5 & : \forall i \in 1 \ldots 10000 : m_b[i+1] \leftarrow m_a[i] \land b[i] \\
R_6 & : \forall i \in 1 \ldots 10000 : m_c[i] \leftarrow m_a[i] \land b[i] \land b[i+1] \\
R_7 & : \forall i \in 1 \ldots 10000 : m_e[i] \leftarrow m_c[i]
\end{align*}

Say that we are interested in finding out whether \(a[1]\) and \(a[10000]\) are true. We set \(m_a[1]\) and \(m_a[10000]\) as true, and then let the bottom-up grounding algorithm run. As a result, we get the following ground program (the rule instantiation information is also given), let us look at the part generated by the query atom \(m_a[1]\) first:

\begin{align*}
m_a[1] & \quad \% \text{given} \\
m_b[1] & \leftarrow m_a[1] \quad \% R_4 \text{ with } i = 1 \\
b[1] & \leftarrow m_b[1] \quad \% R_2 \text{ with } i = 1 \\
m_b[2] & \leftarrow m_a[1] \land b[1] \quad \% R_5 \text{ with } i = 1 \\
b[2] & \leftarrow m_b[2] \quad \% R_2 \text{ with } i = 2 \\
m_c[1] & \leftarrow m_a[1] \land b[1] \land b[2] \quad \% R_6 \text{ with } i = 1 \\
m_e[1] & \leftarrow m_c[1] \quad \% R_7 \text{ with } i = 1 \\
c[1] & \leftarrow m_c[1] \land \neg e[1] \quad \% R_3 \text{ with } i = 1 \\
a[1] & \leftarrow m_a[1] \land b[1] \land b[2] \land \neg c[1] \quad \% R_1 \text{ with } i = 1
\end{align*}

For \(m_a[10000]\), we get the following ground program:

\begin{align*}
m_a[10000] & \quad \% \text{given} \\
m_b[10000] & \leftarrow m_a[10000] \quad \% R_4 \text{ with } i = 10000 \\
b[10000] & \leftarrow m_b[10000] \quad \% R_2 \text{ with } i = 10000 \\
m_b[10001] & \leftarrow m_a[10000] \land b[10000] \quad \% R_5 \text{ with } i = 10000
\end{align*}
Since rule \( R_2 \) is only defined for \( i \in 1 \ldots 10000 \), we cannot ground it to generate the head \( b[10001] \). Therefore, after filtering out the magic rules and variables, our final ground program is:

\[
\begin{align*}
b[1] \\
b[2] \\
c[1] &\leftarrow \neg e[1] \\
\end{align*}
\]

8.4.2 Magic Set Transformation For BFASP

Now we describe how we can generalize the magic set rewriting technique for BFASP. We first describe it for stratified BFASPs (see Sections 6.3 and 8.1), and then explain how we deal with unstratified BFASPs in Section 8.4.3.

Let us first define the query of a BFASP. To build the query \( Q \) for a BFASP \( P \), we ground all its constraints and its objective function, and put all the variables that appear in them in \( Q \). Technically if the problem has output variables, whose value will be printed, they too need to be added to \( Q \). Note that our query does not have any free variables and only contains ground variables.

As described previously, the purpose of the magic set technique is to simulate a top-down computation through bottom-up grounding. For every variable \( a \) in the original program, we create a magic variable \( m.a \) that represents whether we care about \( a \). Additionally, there are magic rules that specify when a magic variable should be created, meaning, when we become interested in the corresponding variable. Consider a simple rule \((a \geq b + c + d, a)\). Let us say that \( u/jb \) of all four variables is equal to \(-\infty\), and we are interested in computing the final value of \( a \). We model this as initially setting \( m.a \) to true. To capture that the value of \( b \) is required to compute the value of \( a \), we add a magic rule \( m.b \leftarrow m.a \). We can have a similar rule for \( m.c \), but actually, we can make the condition for deriving \( m.c \) tighter. If \( b \) can never go higher than \(-\infty\), then there is no need to know the value of \( c \) since the rule is useless until \( b \) is created. Similarly, we are only interested in \( d \) if both \( b \) and \( c \) are created. Therefore, we add the rules: \( m.c \leftarrow m.a \land cr(b) \) and
We already have the necessary conditions in the form of \( \phi_r \) that should be satisfied before a non-ground rule can be instantiated to a useful ground rule. We can utilize that information for the generation of magic rules. Recall that after evaluating the initial conditions, \( \phi_r \) reduces to true, false, a conjunction or a disjunction. For a conjunction, the magic rules are the same as they are for a normal rule in the original magic set technique.

For example, for the above rule \( r = (a \geq b + c + d, a) \), \( \phi_r = cr(b) \land cr(c) \land cr(d) \), as described, we get the magic rules \( m_b \leftarrow m_a \) and \( m_c \leftarrow m_a \land cr(b) \) and \( m_d \leftarrow m_a \land cr(b) \land cr(c) \). For a disjunction, the magic rules are even simpler. For every \( cr(x) \) in the disjunction, we create the magic rule \( m_x \leftarrow m_a \). Note that not all variables in the original rule appear in the condition; some might get removed in the simplification or not be included in the original condition at all. We can ignore them for grounding, but we are interested in their values as soon as we know that the rule can be useful. Therefore, as soon as the magic variable for the head is created, and \( \phi_r \) is satisfied, we are interested in all the variables in the rule that do not appear in \( \phi_r \). Finally, we define the modification step for a rule \( r = (y \geq f(\bar{x}), y) \), written \( \text{modify}(r) \), as changing it to \( r = (y \geq f(\bar{x}) \leftarrow m_y, y) \). The pseudo-code for generation of magic rules and modification of the original rule is given as the function \( \text{magic} \) that takes a rule as its input. It adds magic rules for a rule to a set \( P \). The first two if conditions handle the disjunctive and conjunctive case respectively. The \( \text{for} \) loop that follows generates magic rules for variables that are not in \( \phi_r \). The last line is the modification of the original rule.

The entire bottom-up calculation with magic sets is as follows. First, create magic variables for all the variables in the program and call \( \text{magic} \) for every rule in the program. If the magic rules generated and/or the original rule after modification are not primitive expressions, flatten them. Then, call \( \text{ground} \) on the resulting program. While grounding the constraints, build the query by including \( m_v \) in \( Q \) for every ground variable \( v \) that is in some ground constraint. After grounding, filter all the magic variables from \( Q \), and magic rules from \( R \).

The next example demonstrates the complete bottom-up calculation with magic sets.
Example 8.10. Consider a BFASP with the following rules:

\begin{align*}
R_1 \; \forall i \in [2, 30] \text{ where } i \mod 2 = 0 : \; (a[i] \geq b[i - 1] + y[i], a[i]) \\
R_2 \; \forall i \in [2, 30] \text{ where } i \mod 2 = 0 : \; (y[i] \geq \max(c[2i], d[i + 1]), y[i]) \\
R_3 \; \forall i \in [1, 10] : \; (c[i] \geq 10 \leftarrow s_1[i], c[i]) \\
R_4 \; \forall i \in [1, 10] : \; (b[i] \geq s_2[i + 1], b[i])
\end{align*}

where \(a, b, c, d, y\) are arrays of founded integers with \(u\text{j}b \text{ of } -\infty\), \(s_2\) is an array of standard Booleans and \(s_1\) is an array of standard integers with domains \((-\infty, \infty)\), and the index set of all arrays is equal to \([1, 100]\). Let us compute \(\phi_r\) for each rule. \(\phi_{R_1} = cr(b[i - 1]) \land cr(y[i]), \phi_{R_2} = cr(c[2i]) \lor cr(d[i + 1]),\) and \(\phi_{R_3} = \phi_{R_4} = true\). We get the following magic rules:
8.4 Magic Set Transformation

Let us say we are given the constraint: \(a[2] + a[5] \geq 10\). Processing this, we initialize \(Q\) with the set \(\{m.a[2], m.a[5]\}\). Running ground procedure extends \(Q\) with the following variables, the rule used to derived a variable is given in brackets: \(m.b[1](M1)\), \(m.s_2[2](M6)\), \(b[1](R4)\), \(m.y[2](M2)\), \(m.c[4](M3)\), \(m.d[3](M4)\), \(c[4](R3)\), \(m.s_1[4](M5)\), \(y[2](R2)\), \(a[2](R1)\).

Filtering magic rules, the following ground rules are generated during the grounding (the ujb of variables that are not created are plugged in as constants in rules where they appear): 
\[
(a[2] \leftarrow b[1] + y[2], a[2]), \quad (y[2] \geq \max(c[4], -\infty), y[2]), \quad (c[4] \geq 10 \leftarrow s_1[4], c[4]) \text{ and } (b[1] \geq s_2[2], b[1]).
\]

Now let us roughly compare the different approaches. Without magic sets transformation and only bottom-up grounding, both \(R3\) and \(R4\) yield 10 ground rules each, \(R2\) gives 2 ground rules (for \(i \in \{2, 4\}\)), and \(R1\) gives 2 ground rules as well (for \(i \in \{2, 4\}\)). With exhaustive grounding, the number of rules from \(R1\) to \(R4\) is 48 (14+14+10+10). The most important point to note is that increasing the range of index variables in the generators of rules affects both bottom-up and exhaustive grounding (grnd(\(P\))), but has no effect on grounding with magic sets.

8.4.3 Unstratified BFASPs

If a given BFASP program is unstratified, then the algorithm described above is not sound. There might be parts of the program that are unreachable from the founded atoms appearing in the query but are inconsistent. A simple example is a program with a rule \(a \leftarrow \neg a\) and a constraint \(\neg b\). There are no stable models of this program particularly due
to the first rule which can never be satisfied, but the magic set grounding algorithm will ignore the rule since it is not reachable from the $b$, and therefore wrongly declare $¬b$ as a stable model of the program. This is unconditional inconsistency (Faber et al., 2007). On the other hand, a simple example of conditional inconsistency is the following program with three rules: $p ← ¬q; q ← ¬p; y ← ¬y, ¬q$ and a constraint $¬p ∨ ¬q$. Since we are only interested in $p$ and $q$, our algorithm will ignore the third rule which means that both $p = true, q = false$ and $q = true, p = false$ are stable assignments, but that is clearly wrong. The second assignment is not stable due to the third rule that we ignored. This is conditional dependency which means that some but not all stable models of the restricted ground program are actual stable models of the complete ground program. The source of both types of inconsistencies is unstratified negation as we can prove that without it, we can safely ignore parts of the program that are not reachable from the constraints. More details can be found in the work by Faber et al. (2007).

We overcome this by including in the query all ground magic variables of all array variables that are part of a component in the dependency graph in which there is some increasing edge between any two of its nodes. The following result establishes correctness of our approach.

**Theorem 8.2.** Given a BFASP $P$, let $G$ be equal to $grnd(P)$ and $M$ be a ground BFASP produced by running the magic set transformation after including the unstratified parts of the program in the initial query for a given non-ground BFASP $P$. The stable solutions of $G$ restricted to the variables $vars(M)$ are equivalent to the stable solutions of $M$. That is, if $θ'$ is a stable solution of $G$, then $θ'|vars(M)$ is a stable solution of $M$ and if $θ$ is a stable solution of $M$, then there exists $θ'$ s.t. $θ'$ is a stable solution of $G$ and $θ'|vars(M) = θ$.

**Proof.** Let us first argue about the correctness of our grounding approach presented in Section 8.3. We can analyze each row in Table 8.1 and reason that until the condition is satisfied, the rule can be ignored without changing the stable solutions of the program. We only provide a brief sketch and do not analyze each case in the table. Say, e.g. for $y ≥ max(x_1, \ldots, x_n)$, if the condition is not satisfied, this means that no $x_i$ has a rule in the program that justifies a value higher than its $ujb_i$, and no $x_i$ initially justifies a bound on $y$ that is greater than $ujb(y)$. If we include a ground version of this rule in the program,
then after taking the reduct w.r.t. any assignment, the rule can never justify any bound on the head, and hence can safely be eliminated.

Let $P_i$ be part of $grad(P)$ that is not included in $M$. It can be seen from the description of magic set transformation that any variable in $P_i$ either cannot be reached from any variable in $M$ in the dependency graph of $P$, or can only be reached through useless rules. Since useless rules can be eliminated as argued above, we conclude that no variable in $M$ can reach any variable in $P_i$ in the dependency graph. This obviously also holds for dependency graph of respective reduced program w.r.t. some assignment. This means that for a given assignment $\theta'$, the minimal order computation can first be performed on $M^{\theta'}$ which fixes all the variables in $vars(M)$, and then on $P_{i}^{\theta'}$ which fixes all the remaining variables, i.e., variables in $vars(P) - vars(M)$. Combining both the minimal solutions would be the same as computing the minimal solution for $G^{\theta'}$. This proves the first result.

For the second result, since all unstratified parts in $P$ are included in $M$, all the intra-component edges in the dependency graph of $P_i$ are marked increasing (positive). It can be shown that for such a program, once we fix all the standard variables appearing in any rule in $P_i$, there is a unique stable solution that can be computed as the iterated least fixpoint of $P_i$. This is similar to the well known result for logic programs that states that for a stratified program, the unique stable solution can be computed as the iterated least fixpoint of the program (Corollary 2 by Gelfond and Lifschitz (1988a)). Therefore, if we are given a stable solution $\theta$ for $M$, we can extend it to $\theta'$ by fixing all the unfixed standard variables to any value, and then computing the iterated least fixpoint, which will extend $\theta'$ over founded variables of $P_i$, and will be a unique stable solution given the values of all standard variables.

\section{8.5 Experiments}

We show the benefits of bottom-up grounding and magic sets for computing with BFASPs on the benchmarks described in Section 6.7: RoadCon, UtilPol and CompanyCon.\textsuperscript{26} All experiments were performed on a machine running Ubuntu 12.04.1 LTS with 8 GB of

\textsuperscript{26}All problem encodings and instances can be found at: \url{http://people.eng.unimelb.edu.au/pstuckey/bound_founded}
physical memory and Intel(R) Core(TM) i7-2600 3.4 GHz processor. Our implementation extends MiniZinc 2.0 (LIBMZN) and uses the solver CHUFFED extended with founded variables and rules as described in Chapter 7. Each time in the tables is the median time in seconds of 10 different instances.

Table 8.2 shows the results for RoadCon. \( N \) is the number of nodes, and SCCs is the minimum number of strongly connected components in the graph. We compare exhaustive grounding (simply creating \( grnd(P) \)) against bottom-up grounding, and bottom-up grounding with magic set transformation. A — represents either the flattener/solver did not finish in 10 minutes or that it ran out of memory. Using bottom-up grounding, the

<table>
<thead>
<tr>
<th>( N )</th>
<th>SCCs</th>
<th>Exhaustive Flat</th>
<th>Exhaustive Solve</th>
<th>Bottom-up Flat</th>
<th>Bottom-up Solve</th>
<th>Magic Flat</th>
<th>Magic Solve</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5</td>
<td>4.25</td>
<td>3.34</td>
<td>1.37</td>
<td>.64</td>
<td>.27</td>
<td>.04</td>
</tr>
<tr>
<td>300</td>
<td>15</td>
<td>39.02</td>
<td>—</td>
<td>4.19</td>
<td>1.25</td>
<td>.41</td>
<td>.07</td>
</tr>
<tr>
<td>600</td>
<td>20</td>
<td>237.97</td>
<td>—</td>
<td>19.70</td>
<td>22.56</td>
<td>.83</td>
<td>.96</td>
</tr>
<tr>
<td>900</td>
<td>30</td>
<td>—</td>
<td>—</td>
<td>30.44</td>
<td>127.90</td>
<td>1.17</td>
<td>4.74</td>
</tr>
<tr>
<td>1400</td>
<td>45</td>
<td>—</td>
<td>—</td>
<td>56.99</td>
<td>398.29</td>
<td>1.79</td>
<td>25.66</td>
</tr>
</tbody>
</table>

Table 8.3: UtilPol
founded variables representing shortest paths between two nodes that are not in the same SCC and the corresponding useless rules are not created. Clearly bottom-up grounding is far superior to naively grounding everything, and magic sets substantially improves on this. Tables 8.3 and 8.4 show the results for utilitarian policies and company controls respectively. The running time for exhaustive and bottom-up for these benchmark are similar, therefore, the comparison is only given for bottom-up vs. magic sets. For UtilPol, \( C \) and \( P \) represent the number of citizens and policies respectively, \( C_r \) represents the maximum number of relevant citizens on which the happiness of \( t \) directly or indirectly depends and \( P_r \) is the maximum number of policies on which the happiness of \( t \) and other citizens in \( C_r \) depends. This is the part of the instance that is relevant to the query and the rest is ignored when magic sets are enabled. It can be seen that magic sets outperform regular bottom-up grounding, especially when the relevant part of the instance is small compared to the entire instance. Note that when \( P_r \) is small, the flattening time for magic sets is greater than the solving time since the resulting set of rules is actually simpler. This changes, however, as \( P_r \) is increased. For CompanyCon, \( C \) is the number of total companies while \( C_r \) is the maximum number of companies reachable from the destination in the given ownership graph. The table shows that if \( C_r \) is small compared to \( C \), magic sets can give significant advantages. The unnecessary founded variables and rules can make solving time considerably higher if magic sets optimization

<table>
<thead>
<tr>
<th>Instance</th>
<th>Bottom-up</th>
<th>Magic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Flat</td>
<td>Solve</td>
</tr>
<tr>
<td>1000</td>
<td>24.27</td>
<td>5.20</td>
</tr>
<tr>
<td>1500</td>
<td>53.66</td>
<td>17.52</td>
</tr>
<tr>
<td>2000</td>
<td>94.38</td>
<td>66.81</td>
</tr>
<tr>
<td>3000</td>
<td>209.70</td>
<td>86.35</td>
</tr>
<tr>
<td>3500</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>5000</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 8.4: CompanyCon
is not used.

8.6 Conclusion

Bound Founded Answer Set Programming extends ASP to disallow circular reasoning over numeric entities. While the semantics of BFASP is a simple generalization of the semantics of ASP, to be practically useful we must be able to model non-ground BFASPs in a high level way. In this chapter, we show how we can flatten and ground a non-ground BFASP while preserving its semantics, thus creating an executable specification of the BFASP problem. We show that using bottom-up grounding and magic sets transformation we can significantly improve the efficiency of computing BFASPs. The existing magic set techniques are only defined for the normal rule form, involving only founded Boolean variables. We have extended magic sets to BFASP, a formalism that has significantly more sophisticated rule forms and has both standard and founded variables, that can moreover be Boolean or numeric. To the best of our knowledge, this is the first work of its kind that extends magic set transformation to hybrid systems involving both standard and founded variables such as CLINGCON (Gebser et al., 2009c) and the IDP system (Wittocx et al., 2008).
Part II

Stable Model Counting
Chapter 9
Stable Model Counting and Probabilistic Logic Programming

In this chapter, we look at the problem of stable model counting and how it can be applied to solving inference tasks in probabilistic logic programming. For this chapter, the set of variables \( \mathcal{V} \) is equal to the set of Boolean variables \( \mathcal{V}_B \), and it includes both standard and founded variables.

9.1 Introduction

Consider the counting version of graph reachability problem: given a directed graph, count the number of subgraphs in which node \( t \) is reachable from node \( s \) (Valiant, 1979). This problem can be naturally modeled as a logic program under stable model semantics (Gelfond and Lifschitz, 1988b). Let us say that the input is given by two predicates: \( \text{node}(X) \) and \( \text{edge}(X, Y) \). For each node, we can introduce a decision variable \( \text{in} \) that models whether the node is in the subgraph. Furthermore, we can model reachability \( \text{reach} \) from \( s \) as an inductive definition using the following two rules: 

\[
\text{reach}(s) \leftarrow \text{in}(s)
\]

\[
\text{reach}(Y) \leftarrow \text{in}(Y), \text{reach}(X), \text{edge}(X, Y).
\]

The first rule says that \( s \) itself is reachable if it is in the subgraph. The second rule is the inductive case, specifying that a node \( Y \) is reachable if it is in the subgraph and there is a reachable node \( X \) that has an edge to it. Additionally, say there are arbitrary constraints in our problem, e.g., only consider subgraphs where a certain \( y \) is also reachable from \( s \) etc. This can be done using the integrity constraint: \( \leftarrow \neg \text{reach}(y) \). The number of stable models of this program is equal to the number of solutions of the problem.

There are at least two approaches to counting the stable models of a logic program.
The first is to translate a given logic program to a propositional theory such that there is a one-to-one correspondence between the propositional models of the translated program and the stable models of the original program, and use SAT model counting (Gomes et al., 2009). We show that this approach does not scale well in practice as well as the one that we propose. The second approach is to use an answer set programming (ASP) solver like CLASP (Gebser et al., 2007) or DLV (Leone et al., 2006) and enumerate all models. This approach is extremely inefficient since model counting algorithms have several optimizations like caching and dynamic decomposition that are not present in ASP solvers. This motivates us to build a stable model counter that can take advantage of state-of-the-art ASP technology which combines partial translation and lazy unfounded set (Van Gelder et al., 1988) detection. However, we first show that it is not correct to naively combine partial translation and lazy unfounded set detection with SAT model counters due to the aforementioned optimizations in model counters. We then suggest two approaches to properly integrate unfounded set propagation in a model counter.

We show that we can apply our algorithms to solve probabilistic logic programs (Raedt and Kimmig, 2013). Consider the probabilistic version of the above problem, also called the graph reliability problem (Arora and Barak, 2009). In this version, each node can be in the subgraph with a certain probability $1 - p$, or equivalently, fail with the probability $p$. We can model this by simply attaching probabilities to the $in$ variables. We can model observed evidence as constraints. E.g., if we have evidence that a certain node $y$ is reachable from $s$, then we can model this as the unary constraint (not rule): $reach(y)$. The goal of the problem is to calculate the probability of node $t$ being reachable from node $s$ given the evidence. The probabilistic logic programming solver PROBLOG2 (Fierens et al., 2011) approaches this inference task by reducing it to weighted model counting of the translated propositional theory of the original logic program. We extend PROBLOG2 to use our implementation of stable model counting on the original logic program and show that our approach is more scalable.

We extend the definition of residual programs for ASP. Given an assignment $\theta$ and a set of rules $R$, the residual rules $R|_\theta$ are defined similarly to residual clauses by treating every rule as its logically equivalent clause.
9.2 SAT-Based Stable Model Counting

The most straightforward approach to counting the stable models of a logic program is to translate the program into propositional theory and use a propositional model counter. As long as the translation produces a one-to-one correspondence between the stable models of the program and the solutions of the translated program, we get the right stable model count. Unfortunately, this is not a very scalable approach. Translations based on adding loop formulas (Lin and Zhao, 2004) or the proof-based translation used in PROBLOG2 (Fierens et al., 2011) require the addition of an exponential number of clauses in general (Lifschitz and Razborov, 2006, Vlasselaer et al., 2014). There is also a polynomial sized translation based on level rankings as presented by Janhunen (2004), which we compare against.

Current state of the art SAT-based ASP solvers do not rely on a full translation to SAT. Instead, they rely on lazy unfounded set detection. In such solvers, only the rules are translated to SAT. There is an extra component in the solver which detects unfounded sets and lazily adds the corresponding loop formulas to the program as required (Gebser et al., 2012b). Such an approach is much more scalable for solving ASP problems. However, it cannot be naively combined with a standard SAT model counter algorithm. This is because the SAT model counter requires the entire Boolean formula to be available so that it can check if all clauses are satisfied to calculate the residual program. However, in this case, the loop formulas are being lazily generated and many of them are not yet available to the model counter. Naively combining the two can give the wrong results, as illustrated in the next example.

Example 9.1. Consider a program \( P_1 \) with founded variables \( \{a, b\} \), standard variables \( \{s\} \) and rules: \( a \leftarrow b, b \leftarrow a, a \leftarrow s \). There are only two stable models of the program \( \{a, b, s\} \) and

---

27We remind the reader that this chapter has been presented in the following paper:


The claim in the above paper that this translation does not produce solutions that are in one-to-one correspondence with the stable models of the original program, is wrong. The translation, indeed, is bijective and does preserve the stable model count. Our experiments show that the approach presented in this chapter is still a better approach for our intended benchmarks. See the experiments section for more details.
\{-a, \neg b, \neg s\}. If our partial assignment is \{a, b\}, then the residual program contains an empty theory which means that the number of solutions extending this assignment is 2 (or \(2^{|\{s\}|}\)). This is clearly wrong, since \{a, b, \neg s\} is not a stable model of the program.

Now consider \(P_2\) which is equal to \(P_1\) with these additions: founded variable \(c\), standard variables \{\(t, u\)\} and two rules: \(c \leftarrow a \land t\) and \(b \leftarrow u\). Consider the partial assignment \{\(u, a, b, s\)\}, the residual program has only one rule: \(c \leftarrow t\). It has two stable models, \{\(c, t\)\} and \{\neg c, \neg t\}. Now, with the partial assignment \{\neg u, a, b\}, we get the same residual program and the number of solutions should be: \(2 \times 2^{\{|s|\}} = 4\) which is wrong since \(s\) cannot be false in order for \(a, b\) to be true when \(u\) is false, i.e., \{\neg u, c, t, a, b, \neg s\} and \{\neg u, \neg c, \neg t, a, b, \neg s\} are not stable models of \(P_2\).

In order to create a stable model counter which can take advantage of the scalability of lazy unfounded set detection, we need to do two things: 1) identify the conditions for which the ASP program is fully satisfied and thus we have found a cube of stable models, 2) identify what the residual of an ASP program is so that we can take advantage of caching and dynamic decomposition.

### 9.2.1 Searching On Standard Variables For Stratified Programs

The first strategy is simply to restrict the search to standard variables. If the program is stratified, then the founded variables of the program are functionally defined by the standard variables of the program. Once the standard variables are fixed, all the founded variables are fixed through propagation (unit propagation on rules and the unfounded set propagation). It is important in this approach that the propagation on the founded variables is only carried out on the rules of the program, and not the constraints. Constraints involving founded variables should only be checked once the founded variables are fixed. The reader can verify that in Example 9.1, if we decide on standard variables first, then none of the problems occur. E.g., in \(P_1\), if \(s\) is fixed to either true or false, then we do not get any wrong stable model cubes. Similarly, in \(P_2\), if we replace the second assignment with \{\neg u, s\} which propagates \{\(a, b\)\}, we still get the same residual program, but in this case, it is correct to use the cached value. Note that stratification is a requirement for all probabilistic logic programs under the distribution semantics (Sato, 1995). For such
programs given an assignment to standard variables, the well-founded model of the resulting program is the unique stable model. Furthermore, even in ASP, once we add standard variables in our formalism, stratification over founded variables is not really a limitation as we can efficiently remove all unstratified negation using the transformation described by Niemelä (2008).

9.2.2 Modifying The Residual Program

In ASP solving, it is often very useful to make decisions on founded variables as it can significantly prune the search space. For this reason, we present a more novel approach to overcome the problem demonstrated in Example 9.1.

The root problem in Example 9.1 in both cases is the failure to distinguish between a founded variable being true and being justified, i.e., can be inferred to be true from the rules and current standard and negative literals. In the example, in \( P_1 \), \( a \) and \( b \) are made true by search (and possibly propagation) but they are not justified as they do not necessarily have externally supporting rules (they are not true under stable model semantics if we set \( \neg s \)). In ASP solvers, this is not a problem since the existing unfounded set detection algorithms guarantee that in complete assignments, a variable being true implies that it is justified. This is not valid for partial assignments, which we need for counting stable model cubes. Next, we show that if we define the residual rules (not constraints) of a program in terms of justified subset of an assignment, then we can leverage a propositional model counter augmented with unfounded set detection to correctly compute stable model cubes of a program. In order to formalize and prove this, we need further definitions.

Definition 9.1. Given a program \( P = (V, R, C) \) and a partial assignment \( \theta \), the justified assignment \( JA(P, \theta) \) is the subset of \( \theta \) that includes all standard and founded negative literals plus all the positive founded literals implied by them using the rules of the program. More formally, let \( J_0(\theta) = \theta^- \cup \{ v \in \theta | v \in V_S \} \). Then, \( JA(P, \theta) = J_0(\theta) \cup \{ v \in V_F | v \in \theta, v \in \text{Least}(R|_{J_0(\theta)}) \} \).

**Definition 9.1.** Given a program \( P = (V, R, C) \) and a partial assignment \( \theta \), let \( J = JA(P, \theta) \) and \( U = \text{vars}(\theta) \setminus \text{vars}(J) \). The justified residual program of \( P \), w.r.t. \( \theta \) is written \( P|_\theta \) and is
equal to \((W, S, D)\) where \(S = R|_I, D = C|_\theta \cup \{u| u \in U\}\) and \(W = \text{vars}(S) \cup \text{vars}(D)\).

**Example 9.2.** Consider a program \(P\) with founded variables \(\{a, b, c, d, e, f\}\), standard variables \(\{s, t, u, x, y, z\}\) and the following rules and constraints:

\[
\begin{align*}
a &\leftarrow b. & c &\leftarrow d. & e &\leftarrow \neg f. & \neg s \lor \neg t \\
b &\leftarrow a. & d &\leftarrow u. & f &\leftarrow \neg e. & a \lor b \\
a &\leftarrow s. & f &\lor x \\
b &\leftarrow t.
\end{align*}
\]

Let \(\theta = \{a, b, d, u, \neg e, c, f\}\). Then, \(J_0(\theta) = \{u, \neg e\}\) and \(J_A(P, \theta) = J_0(\theta) \cup \{d, f, c\}\). The justified residual program w.r.t. \(\theta\) has all the rules in the first column and has the constraints: \(\{\neg s \lor \neg t, a, b\}\).

**Theorem 9.1.** Given an ASP program \(P = (V, R, C)\) and a partial assignment \(\theta\), let \(P|_\theta = (W, S, D)\) be denoted by \(Q\). Let the remaining variables be \(V_r = V \setminus (W \cup \text{vars}(\theta))\) and \(\pi\) be a complete assignment over \(W\). Assume every founded variable for which there is no rule in \(S\) is false in \(\theta\).

1. If \(\pi\) is a stable model of \(Q\), then for any assignment \(\theta_r\) over the remaining variables, \(\theta \cup \pi \cup \theta_r\) is a stable model of \(P\).

2. For a given assignment \(\theta_r\) over remaining variables, if \(\theta \cup \pi \cup \theta_r\) is a stable model of \(P\), then \(\pi\) is a stable model of \(Q\).

**Proof.** Let \(J = J_A(P, \theta)\). Note that there cannot be a founded variable in the remaining variables \(V_r\), since if a founded variable is not true in \(\theta\) and does not have a rule in \(S\), then it must be false in \(\theta\) due to the given assumption.

The key point is to view \(R\) as two separate sets of rules, \(S\) and \(R \setminus S\) and argue that we can treat them separately for the purpose of least models. If there is any assignment that extends \(\theta\), then from \(R \setminus S\), we can safely delete the rules whose bodies intersect with \(W\) or \(\theta_r\), as these rules are redundant since all assignments in \(J_0(\theta)\) are sufficient to imply
the founded literals in $J$. Furthermore, the least assignment of reduct of $R \setminus S$ w.r.t. any assignment that extends $\theta$ will be exactly equal to the founded literals in $J$. Therefore:

$$\text{Least}(R^{R \cup \pi \cup \theta_R}) = \forall_f J \cup \text{Least}(S^{\pi \cup \theta_R})$$

Since $\theta_R$ does not have any founded variables and we just argued that we can delete the rules in $R \setminus S$ that have any variable from $\theta_R$ and $\theta_R$ is completely disjoint from $S$ by definition, we can simplify the above equality to: $\text{Least}(R^{R \cup \pi}) = \forall_f J \cup \text{Least}(S^\pi)$.

1. We are given that $\pi$ is a stable model of $Q$, i.e., $\pi \models S$, $\pi \models D$ and $\text{Least}(S^\pi) = \forall_f \pi$. It is easy to show that this implies that $\theta \cup \pi \models R \cup C$. Moreover, from the above equality, we get $\text{Least}(R^{R \cup \pi}) = \forall_f J \cup \pi$. Since, due to constraints added in $D$, $\theta$ and $\pi$ are consistent on founded variables in $V \setminus \text{vars}(J)$, we get $\text{Least}(R^{R \cup \pi}) = \forall_f \theta \cup \pi$.

It is easy to see that any assignment $\theta_R$ can be used to extend $\theta \cup \pi$ without affecting satisfiability or the least model, which means that $\theta \cup \pi \cup \theta_R$ is a stable model of $P$.

2. We are given that $\theta \cup \pi \cup \theta_R$ is a stable model of $P$. By definition of residual programs, we know that $\theta \models C \setminus D$ and the intersection of variables in $D$ and $\theta$ is empty, which means that if $D$ is non-empty, then $\theta$ is not sufficient to satisfy $D$ which implies that $\pi \models D$ (if $D$ is empty, then trivially $\pi \models D$). A similar argument for the case $\pi \models R|_\theta$ can be made. Given that $\theta$ and $\pi$ are consistent, we can also see that $\pi \models S \setminus R|_\theta$ which means that $\pi \models S$. For least model, from the equality that we discussed previously, we are given that $\theta \cup \pi = \forall_f J \cup \text{Least}(S^\pi)$. Again, since $\theta \cup \pi = \forall_f J \cup \pi$, we can derive that $\pi = \forall_f \text{Least}(S^\pi)$ which means that $\pi$ is a stable model of $Q$.

\[\square\]

**Corollary 9.1.** Let the set of rules and constraints of $Q$ decompose into $k$ ASP programs $Q_1 = (W_1, S_1, D_1), \ldots, Q_k = (W_k, S_k, D_k)$ where $W_i = \text{vars}(S_i) \cup \text{vars}(D_i)$ s.t. for any distinct $i, j$ in $1 \ldots k$, $W_i \cap W_j = \emptyset$. Let the remaining variables be: $V_r = V \setminus (W_1 \cup \ldots \cup W_k \cup \text{vars}(\theta))$ and let $\pi_1, \ldots, \pi_k$ be complete assignments over $W_1, \ldots, W_k$ respectively.
1. If $\pi_1, \ldots, \pi_k$ are stable models of $Q_1, \ldots, Q_k$ resp., then for any assignment $\theta_r$ over the remaining variables, $\theta \cup \pi_1 \cup \ldots \cup \pi_k \cup \theta_r$ is a stable model of $P$.

2. For a given assignment $\theta_r$ over remaining variables, if $\theta \cup \pi_1 \cup \ldots \cup \pi_k \cup \theta_r$ is a stable model of $P$, then $\pi_i$ is a stable model of $Q_i$ for each $i \in 1 \ldots k$.

Proof. Theorem 9.1 says that the justified residual program can be solved in isolation and its results can be combined with the parent program. Furthermore, since all ASP programs $Q_1, \ldots, Q_k$ are completely disjoint, both 1 and 2 follow from the Module Theorem (Theorem 1) as given by Janhunen et al. (2007) which says that two mutually compatible assignments that are stable models of two respective programs can be joined to form a stable model of their union program and conversely, a stable model of the combined program can be split into stable models of individual programs, as long as there are no positive interdependencies between the two programs. Ours is a simple special case of Corollary 1 by Janhunen et al. (2007) where all programs and their sets of variables are completely disjoint.

The first part of Theorem 9.1 shows that we can solve the justified residual program independently (as well as cache the result) and extend any of its stable models to a full stable model by assigning any value to the remaining variables of the original program. The second part of the theorem establishes that any full stable model of the original program is counted since it is an extension of the stable model of the residual program. The corollary tells us that if the justified residual program decomposes into disjoint programs, then we can solve each one of them independently, and multiply their counts to get the count for justified residual program.

Example 9.3. In Example 9.2, the justified residual program has only two stable models: $\pi_1 = \{s, a, b, \neg t\}$ and $\pi_2 = \{t, a, b, \neg s\}$. It can be verified that the only stable assignments extending $\theta$ of $P$ are $\theta \cup \pi_1 \cup \theta_{xyz}$ and $\theta \cup \pi_2 \cup \theta_{xyz}$ where $\theta_{xyz}$ is any assignment on the standard variables $x, y, z$. Therefore, the total number of stable models below $\theta$ is $2 \times 2^{\{x, y, z\}} = 16$.

Now say we have another assignment $\theta' = \{a, b, c, d, u, e, \neg f, x\}$. It can be seen that it produces the same justified residual program as that produced by $\theta$ for which we know the stable...
model count is 2. Furthermore, the set of remaining variables is \( \{y, z\} \). Therefore, the number of stable assignments below \( \theta' \) is \( 2 \times 2^{|\{y, z\}|} = 8 \).

In order to convert a model counter to a stable model counter, we can either modify its calculation of the residual program as suggested by Theorem 9.1, or, we can modify the actual program and use its existing calculation in a way that residual of the modified program correctly models the justified residual program. Let us describe one such approach and prove that it is correct. We post a copy of each founded variable and each rule such that the copy variable only becomes true when the corresponding founded variable is justified. More formally, for each founded variable \( v \), we create a standard variable \( v' \), add the constraint \( \neg v' \lor v \), and for each rule \( a \leftarrow f_1 \land \ldots \land f_n \land s_1 \land \ldots \land s_m \) where each \( f_i \) is a positive founded literal and each \( s_n \) is a standard or negative literal, we add the clause \( a' \lor \neg f_1' \lor \ldots \lor \neg f_n' \lor \neg s_1 \lor \ldots \lor \neg s_m \). Most importantly, we do not allow search to take decisions on any of these introduced copy variables. Let this transformation of a program \( P \) be denoted by \( \text{copy}(P) \).

We now show that it is correct to use the above approach for stable model counting. For the following discussion and results, let \( P = (V, R, C) \) be an ASP program, \( \text{copy}(P) = (W, R, D) \) be denoted by \( Q \). Let \( \pi, \pi_1, \pi_2 \) be assignments over \( W \) and \( \theta, \theta_1, \theta_2 \) be their projections over non-copy variables (\( V \)). Let \( Q|\pi \) (similarly for \( \pi_1, \pi_2 \)) be a shorthand for \( (\text{vars}(R|\theta) \cup \text{vars}(D|\theta), R|\theta, D|\theta) \). The results assume that assignments \( \pi, \pi_1, \pi_2 \) are closed under unit propagation and unfounded set propagation, i.e., both propagators have been run until fixpoint in the solver.

To prove the results, we define a function \( prj \) that takes the copy program \( Q \) and \( \pi \) and maps it to the justified residual program w.r.t. to the projection of that \( \pi \) on non-copy variables and then argue that \( Q|\pi \) correctly models the justified residual program. Formally, \( prj(Q, \pi) \) is an ASP program \( P' = (V', R', C') \) constructed as follows. Add every constraint in \( D|\pi \) that does not have a copy variable in \( C' \). For every constraint \( v' \lor \neg f_1' \lor \ldots \lor \neg f_n' \lor \neg s_1 \lor \ldots \lor \neg s_m \) in \( D|\pi \), add the rule \( v \leftarrow f_1 \land \ldots \land f_n \land s_1 \land \ldots \land s_m \) in \( R' \). Let \( U \) be the set of founded variables \( v \) such that \( v \) is true but \( v' \) is unfixed in \( \pi \). For every \( v \) in \( U \), add the constraint \( v \) in \( C' \). Define \( V' \) as variables of \( R' \) and \( C' \). Proposition 9.1 proves that we cannot miss any stable model of the original program if we use the
copy approach.

**Proposition 9.1.** If \( \pi \) cannot be extended to any stable model of \( Q \), then \( \theta \) cannot be extended to any stable model of \( P \).

**Proof sketch.** Say \( \theta \) has an extension \( E \) that is a stable model of \( P \). We can show that running unit propagation on \( Q \) and \( E \) yields a solution of \( Q \) that is an extension of \( \pi \), which contradicts what is given. \( \square \)

Theorem 9.2 establishes that we can safely use \( Q|_{\pi} \) to emulate the justified residual program \( P|_{\theta} \). Corollary 9.2 says that if we detect a stable model cube of \( Q|_{\pi} \), then we also detect a stable model cube of the same size for the justified residual program. This corollary and Proposition 9.1 prove that the stable model count of the actual program is preserved.

**Theorem 9.2.** \( P|_{\theta} = \text{prj}(Q, \pi) \).

**Proof.** Recall the definition of justified assignment: \( JA(P, \theta) = J_0(\theta) \cup \{v \in \bigvee F|v \in \theta, v \in \text{Least}(R|_{J_0(\theta)})\} \). Also recall that any copy constraint \( r' \) in \( Q \) has the form: \( v' \lor \lnot f_1' \lor \ldots \lor \lnot f_n' \lor \lnot s_1 \lor \ldots \lor \lnot s_m \) and by definition, each \( r' \) is the copy of a rule \( r \) in \( P \), which is \( v \leftarrow f_1 \land \ldots \land f_n \land s_1 \land \ldots \land s_m \). Recall that \( v', f_1', \ldots, f_n' \) are copy variables of \( v, f_1, \ldots, f_n \) respectively, \( f_1, \ldots, f_n \) are positive literals in \( r \) and \( s_1, \ldots, s_m \) are either standard or negative literals in \( r \). We show that the sets of rules, constraints, and variables of \( \text{prj}(Q, \pi) \) and \( P|_{\theta} \) are equal, therefore, they are equal. We begin by reasoning about the sets of rules.

Let us focus on the seed of the justified assignment \( J_0(\theta) \). It is easy to see that \( J_0(\pi) \cap \bigvee = J_0(\theta) \). Since \( r' \) and \( r \) share all standard and negative literals, their residuals w.r.t. these literals will be simplified in exactly the same way, i.e., if \( r'|_{J_0(\pi)} = v' \lor \lnot f_1' \lor \ldots \lor \lnot f_n' \lor \lnot s_1 \lor \ldots \lor \lnot s_m \), then \( r|_{J_0(\theta)} = v \leftarrow f_1 \land \ldots \land f_n \land s_1 \land \ldots \land s_m \). The core point of the proof is that running unit propagation on the set of copy rules \( r'|_{J_0(\pi)} \) is analogous to computing \( \text{Least}(R|_{J_0(\theta)}) \). Each application of unit propagation on \( r' \) that derives \( v' \) must also derive \( v \) in \( r|_{J_0(\theta)} \). This means that if, due to this propagation, the set of copy variables \( J' = \{v_1', \ldots, v_j'\} \) is derived, then \( JA(P, \theta) \setminus J_0(\theta) = \{v_1, \ldots, v_j\} \). Furthermore, since no decisions on copy variables are allowed, and there is no other constraint that
can possibly derive a literal \( v' \), unit propagation cannot derive any other positive copy literal. Now, let us view \( \text{prj}(Q, \pi) \) as individual applications of a function \( \text{prj}_{\text{copy}} \) on each copy rule \( r' \) to produce \( r \). We can see that \( \text{prj}_{\text{copy}}(r|_{A(\pi'):J'}) = r|_{A(P, \theta)} \). Therefore, the set of rules in \( \text{prj}(Q, \pi) \) and \( P|_\theta \) is exactly the same.

From above, it also follows that the set \( U \) in the construction of \( \text{prj}(Q, \pi) \) and the set \( U \) in Definition 9.1 are equal. Since \( \theta \) is just the restriction of \( \pi \) on non-copy variables, the residual of any constraint that has non-copy variables only is the same and since the consistency constraints due to \( U \) are also the same, the set of constraints \( C' \) and \( C \) are equal. Since the rules and constraints are equal in \( \text{prj}(Q, \pi) \) and \( P|_\theta \), their variables are also equal.

**Corollary 9.2.** If \( Q|_{\pi} \) has no rules or constraints and there are \( k \) unfixed variables, then \( \theta \) is a stable model cube of \( P|_{\theta} \) of size \( 2^k \).

**Proof.** Since \( Q|_{\pi} \) is empty, \( \text{prj}(Q, \pi) \) is also empty, and since \( P|_{\theta} = \text{prj}(Q, \pi) \), this means that \( P|_{\theta} \) is a stable model cube. Furthermore, we can show that all founded variables and their copies must be fixed in \( Q|_{\pi} \), which means all \( k \) variables must be standard variables, therefore, the size of stable model cube of \( P|_{\theta} \) is \( 2^k \).  

The next two corollaries prove that the copy approach can be used for caching dynamic decomposition respectively.

**Corollary 9.3.** If \( Q|_{\pi_1} = Q|_{\pi_2} \), then \( P|_{\theta_1} = P|_{\theta_2} \).

**Proof.** Follows directly from Theorem 9.2 and definition of \( \text{prj} \) function.

**Corollary 9.4.** If \( Q|_{\pi} \) decomposes into \( k \) disjoint components \( Q_1, \ldots, Q_k \), then \( P|_{\theta} \) decomposes into \( k \) disjoint components \( P_1, \ldots, P_k \) such that \( P_i = \text{prj}(Q_i, \pi_i) \) where \( \pi_i \) is projection of \( \pi \) on \( \text{vars}(Q_i) \).

**Proof.** Note that the disjointness of components of \( Q|_{\pi} \) is really determined by its constraints, and not its rules. The residual rules are always stronger (have fewer variables) than their respective residual copy constraints, because it is possible that a founded variable \( v \) is true in \( \pi \) but its copy variable \( v' \) is unfixed. The opposite is not possible and
moreover, standard and negative literals are shared in a rule and its corresponding copy constraint. This property of residual rules is important since $\text{prj}(Q, \pi)$ works by projecting each individual copy rule to its original form in $P|_\theta$ and completely ignores the residual set of rules in $Q|_\pi$.

It is clear from definition of $\text{prj}(Q, \pi)$ that the projection of each rule merely replaces the copy variables with their corresponding founded variables. This means that we can take each disjoint component of $Q|_\pi$ and map it to its counterpart in $P|_\theta$ using the projection function. Non-copy constraint translation in $\text{prj}(Q, \pi)$ is also straight-forward and cannot combine multiple disjoint components in $Q|_\pi$ to one component in $P|_\theta$ or split one component in $Q|_\pi$ to multiple components in $P|_\theta$. Finally, addition of the unary clauses for all variables in $U$ (as used in the definition of $\text{prj}(Q, \pi)$) also does not affect the components; by definition, a variable $v$ in $U$ must have at least one copy constraint $v' \lor \neg f'_1 \lor \ldots \neg f'_n \lor \neg sn_1 \lor \ldots \neg sn_m$ which will be projected to $v \leftarrow f_1 \land \ldots f_n \land sn_1 \land \ldots sn_m$ in $P|_\theta$. Adding $v$ as a constraint does not affect the component $P_i$ in which this projected rule appears in $P|_\theta$.

9.3 PROBLOG2 Via Stable Model Counting

In this section, we describe how we apply stable model counting in the probabilistic logic programming solver PROBLOG2 (Fierens et al., 2013). A probabilistic logic program is a collection of mutually independent random variables each of which is annotated with a probability, derived variables, evidence constraints (known truth values of a subset of variables) and rules for the derived variables. The distribution semantics (Sato, 1995) says that for a given assignment over the random variables, the values of the derived variables is given by the well-founded model. Furthermore, the weight of that world is equal to the product of probabilities of values of the random variables. In our setting, it is useful to think of random variables, derived variables, evidence constraints, and rules as standard variables, founded variables, constraints and rules respectively. PROBLOG2 handles various inference tasks, but the focus of this chapter is computing the marginal probability of query atoms given evidence constraints. The probability of a query atom is equal to
the sum of weights of worlds where a query atom and evidence are satisfied divided by
the sum of weights of worlds where the evidence is satisfied.

Figure 9.1 shows the execution of a PROBLOG2 program. The input is a non-ground
probabilistic logic program which is given to the grounder that cleverly instantiates only
parts of the program that are relevant to the query atoms, similar to how magic set transfor-
mation (Bancilhon et al., 1985) achieves the same goal in logic programming. The ground
program and the evidence is then converted to CNF using the proof based encoding that
we discussed earlier. This CNF is passed on to a knowledge compiler like DSHARP (Muise
et al., 2012). DSHARP is an extension of SHARPSAT (Thurley, 2006) where the DPLL-style
search is recorded as d-DNNF. The d-DNNF produced by the knowledge compiler is
given to the parser of PROBLOG2 along with the ground queries and probabilities of the
random variables. The parser evaluates the probability of each query by crawling the
d-DNNF as described in (Fierens et al., 2013).

Our contribution is in the components in the dotted box in Figure 9.1. We have imple-
mented stable model counting by extending the propositional model counter SHARPSAT
as described in the previous section. Since SHARPSAT is part of the knowledge compiler
DSHARP, our extension of SHARPSAT automatically extends DSHARP to a stable model
knowledge compiler. The CNF conversion component in PROBLOG2 chain is replaced
by a simple processing of the ground program and evidence to our desired input format.
In the first approach where the search is restricted to standard variables, the evidence
needs to be passed on to our stable model counter which posts a nogood (the current
assignment of standard variables) each time an evidence atom is violated. In the ap-
proach given in Section 9.2.2, however, we post each evidence as a unit clause, much like PROBLOG2 does in its CNF conversion step. Including evidence in constraints in the second approach is safe since our residual program relies on the justified assignment only, and propagation on founded literals that makes them true due to constraints does not change that. Outside the dotted box in the figure, the rest of the PROBLOG2 logic remains the same.

9.4 Experiments

We compare the two approaches based on implementation of unfounded set detection as explained in Section 9.2 against the proof based encoding of PROBLOG2. We use two well-studied benchmarks: SmokersFriends (Fierens et al., 2011) problem and the graph reliability problem (GraphRel) (Arora and Barak, 2009) with evidence constraints.

In both problems, the graph is probabilistic. In GraphRel, the nodes are associated with probabilities while in SmokersFriends, the edges have probabilities. Naturally, for \( n \) nodes, the number of random variables is in \( O(n) \) and \( O(n^2) \) for GraphRel and SmokersFriends respectively. Due to this, GraphRel has significantly more loops per random variables in the dependency graph which makes it more susceptible to the size problems of eager encoding. We refer to the fixed search approach of Section 9.2.1 as ASPROBLOGS and the proper integration of unfounded set detection through the use of copy variables of Section 9.2.2 as ASPROBLOG. All experiments were run on a machine running Ubuntu 12.04.1 LTS with 8 GB of physical memory and Intel(R) Core(TM) i7-2600 3.4 GHz processor.

Table 9.1 shows the comparison between PROBLOG2, ASPROBLOG and APROBLOGS on GraphRel on random directed graphs. The instance is specified by \( N \), the number of nodes, and \( P \), the probability of an edge between any two nodes. The solvers are compared on the following parameters: time in seconds (Time), number of variables and clauses in the input program of DSHARP (V and C resp.), number of decisions (D), average decision level of backtrack due to conflict or satisfaction (A), the size in megabytes of the d-DNNF produced by DSHARP (S), and for APROBLOG and APROBLOGS, the
## 9.4 Experiments

<table>
<thead>
<tr>
<th>Instance</th>
<th>PROBLOG2</th>
<th>ASPROBLOG</th>
<th>ASPROBLOGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>Time</td>
<td>V C D A L S</td>
<td>Time</td>
</tr>
<tr>
<td>N</td>
<td>P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.5</td>
<td>11.33</td>
<td>2214</td>
</tr>
<tr>
<td>11</td>
<td>0.5</td>
<td>115.75</td>
<td>6601</td>
</tr>
<tr>
<td>12</td>
<td>0.5</td>
<td>—</td>
<td>16210</td>
</tr>
<tr>
<td>13</td>
<td>0.5</td>
<td>—</td>
<td>59266</td>
</tr>
<tr>
<td>15</td>
<td>0.5</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>20</td>
<td>0.5</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>25</td>
<td>0.25</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>30</td>
<td>0.1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>31</td>
<td>0.1</td>
<td>—</td>
<td>37992</td>
</tr>
<tr>
<td>32</td>
<td>0.1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>33</td>
<td>0.1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>34</td>
<td>0.1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>35</td>
<td>0.1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>36</td>
<td>0.1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>37</td>
<td>0.1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>38</td>
<td>0.1</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 9.1: Comparison of PROBLOG2, ASPROBLOG, and ASPROBLOGS on the Graph Reliability problem

The number of loops produced during the search ($L$). Each number in the table represents the median value of that parameter from 10 random instances of the size in the row. The median is only defined if there are at least (6) output values. A ‘—’ represents memory exhaustion or a timeout of 5 minutes, whichever occurs first. A ‘—’ in columns 'Time', 'D', 'A', 'L', 'S' means that the solver ran out of memory but the grounding and encoding was done successfully, while a ‘—’ in all columns of a solver means that it never finished encoding the problem. We show the comparison on three types of instances: small graphs with high density, medium graphs with high to medium density, and large graphs with low density.

Clearly ASPROBLOG and ASPROBLOGS are far more scalable than PROBLOG2. While PROBLOG2 requires less search (since it starts with all loop formulae encoded) the overhead of the eager encoding is prohibitive. For all solved instances, ASPROBLOG has...
the best running time and d-DNNF size, illustrating that the search restriction of AS-PROBLOGS degrades performance significantly. While the encoding for ASPROBLOGS is always smallest, the encoding with copy variables and rules of ASPROBLOG is not significantly greater, and yields smaller search trees and fewer loop formulae. It is clearly the superior approach.

Unfortunately, the level ranking or rule based conversion of a logic program under stable model semantics to SAT, is not supported by PROBLOG2 at the time of writing. However, we compared running the model counter SHARPSAT (which is already an improvement over DSHARP) on the level ranking encoding produced by LP2SAT using the same logic program on the same instances. The last instance on which counting succeeds is one where \( N = 15 \). Beyond that, SHARPSAT runs out of memory.

Figure 9.2 compares the performance of PROBLOG2 and ASPROBLOG on Smokers-Friends when the number of random variables is fixed to 31 and the problem size is increased. In the problem description, there are two sets of random variables, the stress and the influences variables. The first one exists for each person in the graph, while the latter exists for every edge in the graph. In our setting, for an instance with \( n \) persons, the number of influences random variables is equal to \( 31 - n \). The rest of the influences variables are fixed to true or false at run time. For the smallest instances of sizes 7 and 8, PROBLOG2 and ASPROBLOG have similar performance. For instances 9 to 12, PROBLOG2
does better than ASPROBLOG where the latter cannot solve instances 11 and 12 due to memory exhaustion. The reason is that the complete encoding in PROBLOG2 propagates better and the extra unfounded set check at each node in the search tree in ASPROBLOG does not pay off. But as the number of people increases and the number of probabilistic edges becomes less, the problem becomes easier for ASPROBLOG but not for PROBLOG2. The reason is that by fixing the probabilistic edges, we are just left with $n$ external rules, and many internal rules, making many founded variables logically equivalent to each other. In the last instance, the number of loop formulas required for the problem is only one! Our lazy approach benefits from this structure in the problem, while PROBLOG2 does not. Our experiments with the same range of instances but with number of random variables fixed to 33 and 35 show similar behaviour of PROBLOG2 and ASPROBLOG where initially, PROBLOG2 does better, followed by hard instances for both, and finally, ASPROBLOG detecting the structure and solving the last few instances in less than 2 seconds.

### 9.5 Conclusion

Stable model counting is required for reasoning about probabilistic logic programs with positive recursion in their rules. We demonstrate that the current approach of translating logic programs eagerly to propositional theories is not scalable because the translation explodes when there is a large number of recursive rules in the ground program. We give two methods to avoid this problem which enables reasoning about significantly bigger probabilistic logic programs.
Chapter 10
Projected Stable Model Counting

In this chapter, we describe several algorithms for counting the models of a propositional theory, projected on a subset of vocabulary that we refer to as priority variables. We then explain how these techniques can be extended to solve projected stable model counting. For the first part when our discussion is restricted to projected model counting, the set of variables is equal to standard Boolean variables only, i.e., $V = V_B$ and $V = V_S$. When we discuss the extension to projected stable model counting of a logic program, $V$ also includes founded variables.

The projected model counting problem can be formally defined as follows. We represent the set of priority variables as $P \subseteq V$. Let the non-priority variables be $N$, i.e., $N = V \setminus P$. Given a cube $\theta'$ of a CNF formula $F$, then $\theta \equiv \theta'_P$ is a projected cube of $F$. The size of the projected cube is equal to $2^{|P| - |\theta|}$. The projected model count of $F$, $\text{count}(F, P)$ is equal to the number of projected cubes of size 1. The projected model count can also be defined as the number of assignments $\theta$ s.t. $\text{vars}(\theta) = P$ and there exists an assignment $\theta'$ s.t. $\text{vars}(\theta') = N$ and $\theta \cup \theta'$ is a solution of $F$. After describing algorithms and experiments related to projected model counting, we formally define projected stable model counting and extensions of those algorithms for it.

10.1 Introduction

Model counting is the task of computing the number of models of a given propositional theory, represented as a set of clauses (SAT). Often, instead of the original model count, we are interested in model count projected on a set of variables $P$.

Given a problem on variables $P$, we may need to introduce additional variables to encode the constraints on the variables $P$ into Boolean clauses in the propositional theory $F$. Counting the models of $F$ will not give the correct count if the new variables are not functionally defined by the original variables $P$. Thankfully most methods of encoding
constraints introduce new variables that are functionally defined by original variables, but there are cases where the most efficient encoding of constraints does not enjoy this property. Hence we should consider projected model counting for these kinds of problems.

Alternatively, in the counting problem itself, we may only be interested in some of the variables involved in the problem. Unless the interesting variables functionally define the uninteresting variables, we need projected model counting. An example is in evaluating robustness of a given solution. The goal is to count the changes that can be made to a subset of variables in the solution such that it still remains a solution (possibly after allowing some repairs, e.g. in supermodels of a propositional theory (Ginsberg et al., 1998)). The variables representing change are priority variables. In our benchmarks, we consider an example from the planning domain, where we are interested in robustness of a given partially ordered plan to the initial conditions, i.e., we want to count the number of initial states, such that the given partially ordered plan still reaches the given goal state(s).

Projected model counting is a challenging problem that has received little attention. It is at least as hard as model counting which is the special case where all variables are priority variables. Projected model counting can be considered a special case of QBF counting with a single level of quantification. There has been little development of specialized algorithms for projected model counting in the literature. To the best of our knowledge, the only dedicated attempts at solving the problem are presented by Klebanov et al. (2013) and Gebser et al. (2009b). In the latter, the primary motivation is solution enumeration, and not counting. There is work on closely related problems such as projection or forgetting in formulas that are in deterministic decomposable negation normal form (d-DNNF) (Darwiche, 2001) while maintaining determinism (Darwiche and Marquis, 2002).

In this chapter, we present three different approaches for projected model counting.

- The first technique is straightforward and its basic idea is to modify DPLL-based model counters to search first on the priority variables, followed by finding only a single solution for the remaining problem. This technique is not novel and has been proposed by Klebanov et al. (2013). It has also been suggested in (Palacios
et al., 2005) in a slightly different context. Unlike (Klebanov et al., 2013) which uses external calls to Minisat to check satisfiability of non-priority components, we handle all computations within the solver.

- The second approach is a significant extension of the algorithm presented by Gebser et al. (2009b). The basic idea is that every time a solution $S$ is found, we generalize it by greedily finding a subset of literals $S'$ that are sufficient to satisfy all clauses of the problem. By adding $\neg S'$ as a clause, we save an exponential amount of search that would visit all extensions of $S'$. This extension conveniently blends in the original algorithm by Gebser et al. (2009b), which has the property that the number of blocking clauses are polynomial in the number of priority variables at any time during the search.

- Our third technique is a novel idea which reuses model counting algorithms: computing the D-DNNF of the original problem, forgetting the non-priority variables in the D-DNNF, converting the resulting DNNF to CNF, and counting the models of this CNF.

We compare these three techniques on different benchmarks to illustrate the various strengths and weaknesses they have.

## 10.2 Projected Model Counting

In this section, we present three techniques for projected model counting.

### 10.2.1 Restricting Search To Priority Variables

This algorithm works by slightly modifying the DPLL-based model counters as follows. First, when solving any component, we only allow search decisions on non-priority variables if the component does not have any priority variables. Second, if we find a cube for a component, then the size of that cube is equal to 2 to the power of number of priority variables in the component. Finally, as soon as we find a cube for a component, we recursively mark all parent components that do not have any priority variables as solved, as
a result, the count of 1 from the last component is propagated to all parent components whose clauses are exclusively on non-priority variables. Essentially, we store the fact that such components are 'satisfiable'.

**Example 10.1.** Consider the following program $F$ with priority variables $p, q, r$ and non-priority variables $x, y, z$.

$$(\neg q \lor x \lor \neg p) \quad (\neg r \lor \neg y \lor z) \quad (r \lor \neg z \lor \neg p)$$

$$(z \lor y \lor \neg p \lor r) \quad (r \lor z \lor \neg y \lor \neg p) \quad (p \lor q)$$

Here is the trace of a possible execution using the algorithm in this subsection. We represent a component as a pair of (unfixed) variables and residual clauses.

1a. Decision $p$. The problem splits into $C_1 = (\{q, x\}, (\neg q, x))$ and $C_2 = (\{r, y, z\}, (\neg r, \neg y, z), (r, \neg z), (z, y, r), (r, z, \neg y))$.

2a. We solve $C_1$ first. Decision $\neg q$. We get $C_3 = (\{x\}, \emptyset)$ and $\text{count}(C_3) = 1$ (trivial), we backtrack to $C_1$. 2b. Decision $q$, propagates $x$, and it is a solution. We backtrack and set $\text{count}(C_1) = \text{count}(C_3) + 1 = 2$.

2c. Now, we solve $C_2$. Decision $r$ gives $C_4 = (\{y, z\}, (\neg y, z))$.

3a. Decision $z$, we get $C_5 = (\{y\}, \emptyset)$ and $\text{count}(C_5) = 1$. We backtrack to level $C_2$ setting $\text{count}(C_4) = 1$ since the last decision was a non-priority variable.

2d. Decision $\neg r$ fails (propagates $z, y, \neg y$). We set $\text{count}(C_2) = \text{count}(C_4) = 1$ and backtrack to root $F$ to try the other branch.

1b. Decision $\neg p$, propagates $q$ and gives $C_6 = (\{x\}, \emptyset)$ and $C_7 = (\{r, y, z\}, (\neg r, \neg y, z))$. We note that $\text{count}(C_6) = 1$ (trivial) and move on to solve $C_7$.

2e. Decision $\neg r$ gives $C_8 = (\{y\}, \emptyset)$ and $C_9 = (\{z\}, \emptyset)$ with counts 1 each. We go back to $C_7$ to try the other branch.

2f. Decision $r$ gives $C_{10} = (\{y, z\}, (\neg y, z))$ which is the same as $C_4$ which has the count of 1. Therefore, $\text{count}(C_7) = \text{count}(C_8) \times \text{count}(C_9) + \text{count}(C_4) = 2$. All components are solved, and there are no more choices to be tried, we go back to root to get the final model count. A visualization of the search is shown in Figure 10.1.

The overall count is $\text{count}(F) = \text{count}(C_1) \times \text{count}(C_2) + \text{count}(C_6) \times \text{count}(C_7) = 4$. □
10.2 Projected Model Counting

10.2.2 Blocking Seen Solutions

This approach extends the projected model counting algorithm given by Gebser et al. (2009b). The algorithm is originally for model enumeration, not model counting, and therefore, it suffers in instances where there are small number of cubes, but the number of extensions of these cubes to solutions is large. We present a modification of the algorithm that does not have this shortcoming. But first, let us briefly summarize the motivation behind the algorithm and its technical details.

The motivation presented by Gebser et al. (2009b) is absence of any specialized algorithm in SAT (as well as ASP) for model enumeration on a projected set of variables, and the obvious flaws in the following two straight-forward approaches for model enumeration. The first is essentially the approach from the previous subsection without dynamic decomposition, caching, and cube detection, i.e., searching on variables in $\mathcal{P}$ first and checking for a satisfying extension over $\mathcal{N}$. This approach, according to the paper, is doomed to fail, although the claim is never substantiated in the experiments. The sec-
ond approach is to keep track of solutions that have been found and for each explored solution $\theta$, add the blocking clause $\neg \theta_P$ (this is also presented by Klebanov et al. (2013), although the algorithm restarts and calls MINISAT by adding the clause each time a solution is found). In the worst case, the number of solutions can be exponential in $|P|$, and this approach, as experiments confirm, can quickly blow up in space. Note that, as opposed to the learned clauses which are redundant w.r.t. the original CNF and can be removed any time during the search, the blocking clauses need to be stored permanently, and cannot be removed naively.

The algorithm of Gebser et al. (2009b) runs in polynomial space and works as follows. At any given time during its execution, the search is divided into controlled and free search. The free part of the search runs as an ordinary modern DPLL-based SAT solver would run with backjumping, conflict-analysis etc. In the controlled part of the search, the decision literals are strictly on variables in $P$ and how they are chosen is described shortly. Following the original convention, let $bl$ represent the last level of controlled search space. Initially, it is equal to 0. Every time a solution $\theta$ (with projection $\theta_P$) is found, the search jumps back to $bl$, selects a literal $x$ from $\theta_P$ that is unfixed, and forces it to be the next decision. It increments $bl$ by 1, adds the blocking clause $\neg \theta_P$ and most importantly, couples the blocking clause with the decision $x$ in the sense that when we backtrack from $x$ and try (force) $\neg x$, $\neg \theta_P$ can be removed from memory as it is satisfied by $\neg x$. Since backtracking in the controlled region does not skip over any solution, all solutions with $x$ will have been explored. Furthermore, with $\neg x$, all subsequent blocking clauses that were added will have been satisfied since all of them include $\neg x$. This removal of clauses ensures that the number of blocking clauses at any given time is in $O(|P|)$.

We now describe how we extend the above algorithm by adding solution minimization to it. We keep a global solution count, initially set to 0. Once a solution $\theta$ is found, we generalize (minimize) the solution as shown in the procedure shrink Figure 10.6. We start constructing the new solution cube $S$ by adding all current decisions from 1 \ldots $bl as well as all the non-priority variables in $S$ (which we project away in the end). Then, as long as the residual w.r.t. $S$ is not empty, we pick a literal $p$ from any one of the clauses
and add it to $S$. As a heuristic, given a clause, we choose $p$ to be the literal with the highest frequency in the original CNF. Once the residual is empty, we have found a solution cube, we project away the non-priority literals and use $\neg S$ as a blocking clause instead of the one generated by the algorithm above ($\neg \theta_P$). Finally, we add $2^{|P| - |S|}$ to the global count. The rest of the algorithm remains the same. Note that the decision literals from the controlled part of the search are necessary to add in the cube, since the algorithm by Gebser et al. (2009b) assumes that once a controlled decision is retracted, all the blocking clauses that were added below it are satisfied. This could be violated by our solution minimization if we do not add controlled decisions to $S$.

Example 10.2. Consider the CNF in Example 10.1. Initially, the controlled search part is empty, $B = \emptyset$ and $bl = 0$ as per the original algorithm. Say CLASP finds the solution: $\{p, \neg q, x, z, r, \neg y\}$. shrink produces the generalized solution: $S = \{r, p\}$ by parsing the clauses $(r \lor z \lor \neg p)$ and $(p \lor q)$ respectively (all other clauses can be satisfied by non-priority literals). We increment the model count by $2 = 2^{|S|}$, store the blocking clause $\neg S \equiv (\neg r \lor \neg p)$ and increment $bl$ by 1. Say, we pick $r$, due to the added blocking clause, it propagates $\neg p$, which

```
shrink(\theta)
S := \theta_N \cup \{\text{dec}(i) : i \in 1 \ldots bl\} \quad \% \text{initializing } S

while ((C \cup B)|S \neq \emptyset)
    Q = \{p \in cl, cl \in (C \cup B)|S\}
    p = Q.pop()
    S.add(p)
    S := S_p
    count := count + 2^{|P| - |S|}
    B.add(\neg S)
```

Figure 10.2: Pseudo-code for shrinking a solution $\theta$ of original clauses $C$ and blocking clauses $B$ to a solution cube $S$, adding its count and a blocking clause to prevent its reoccurrence.
propagates \( q \). Say that CLASP now finds the solution \( \{ r, \neg p, q, \neg y, z, x \} \). In \textit{shrink}, we start by including \( r \) in \( S \) since that is a forced decision, and then while parsing the clauses, we get \( S = \{ r, \neg p, q \} \). Note that if we didn't have to include the blocking clause \((\neg r \lor \neg p)\), then we could get away with \( S = \{ r, q \} \) which would be wrong since that shares the solution \( \{ r, q, p \} \) with the previous cube. We increment the count to 3 and cannot force any other decision, so we try the decision \( \neg r \) in the controlled part. At the same time, upon backtracking, we remove all blocking clauses from \( B \), so it is now empty. Say CLASP finds the solution \( \{ \neg r, \neg p, q, x, \neg y, z \} \), \textit{shrink} gives \( S = \{ \neg r, \neg p, q \} \). We increment the count to 4, and when we add \( \neg S \) as a blocking clause, there are no more solutions under \( \neg r \). Therefore, our final count is 4.

Figure 10.3: A visualization of counting models via blocking solutions. The curly arcs indicate free search, ending in a solution, with an associated count. The controlled search is indicated by full arcs, and blocking clauses associated with controlled search decisions are shown on arcs.

### 10.2.3 Counting Models Of Projected d-DNNF

As mentioned in Section 2.8, it is possible to do model counting on d-DNNF in polynomial time (in the size of the d-DNNF), however, once we perform projection on \( P \) (or forgetting on \( N \) (Darwiche and Marquis, 2002)) by replacing all literals whose variables are in \( N \) with \textit{true}, the resulting logical formula is not deterministic anymore and model counting is no longer tractable (Darwiche and Marquis, 2002).

In this approach, we first compute the d-DNNF of \( F \), then project away the literals from the d-DNNF whose variables are in \( N \), convert this projected DNNF back to CNF, and then count the models of this CNF. The pseudo-code is given in Figure 10.5. The
conversion from d-DNNF to CNF is formalized in the procedure \texttt{d2c}, which takes as its input a d-DNNF (as a list of nodes \texttt{Nodes}) and returns a CNF \texttt{C}. It is assumed that \texttt{Nodes} is topologically sorted, i.e., the children of all nodes appear before their parents. \texttt{d2c} maps nodes to literals in the output CNF with the dictionary \texttt{litAtNode}. It also maps introduced (Tseitin) variables to expressions that they represent in a map \texttt{litWithHash}. \texttt{v} represents the index of the next Tseitin variable to be created. \texttt{d2c} initializes its variables with the method \texttt{init()}. Next, it visits each node \texttt{n}, and checks its type. If it is a literal and if it is a non-priority variable, then it is replaced with \texttt{true} (projected away), otherwise, the node is simply mapped to the literal. If \texttt{n} is an AND or an OR node, then we get corresponding literals of its children from the method \texttt{simplify}. We compute the hash to see if we can reuse some previous introduced variable instead of introducing a new one. If not, then we create a new variable through the method \texttt{Tseitin} which also posts the corresponding equivalence clauses in \texttt{C}. Finally, we post a clause that says that the literal for the root (which is the last node) should be true. The method \texttt{simplify} essentially maps all the children nodes to their literals. Furthermore, if one of the literals is \texttt{true} and the input is an OR-node, it returns a list containing a true literal. For an AND node, it filters all the true literals from the children.

The next theorem shows that the method described in this section for projected model counting is correct.

\textbf{Theorem 10.1.} \textit{count}(C) = \textit{count}(F, \mathcal{P})

\textit{Proof.} The entire algorithm transforms the theory from \textit{F} to \textit{C} by producing 2 auxiliary states: the d-DNNF of \textit{F} (let us call it \textit{D}) and the projection of this d-DNNF (let us call this \textit{D}_P). By definition, \textit{F} and \textit{D} are logically equivalent. On the other end, notice that the models of \textit{D}_P and \textit{C} are in one-to-one correspondence. Although the two are not logically equivalent due to the addition of Tseitin variables, it can be shown that these variables do not introduce any extra model nor eliminate any existing model since they are simply functional definitions of variables in \texttt{P} by construction (as a side note, the only reason for introducing these variables is to efficiently encode \textit{D}_P as CNF, otherwise, \textit{C} and \textit{D}_P would be logically equivalent). Furthermore, we can show that the simplifications (replacing \texttt{true} \lor E with \texttt{true} and \texttt{true} \land E with \texttt{E}) in the procedure \texttt{simplify}, and
reusing Tseitin variables (through hashing) also do not affect the bijection. This just leaves us with the task of establishing bijection between the models of $D$ and $D_P$, which, fortunately, has already been done by Darwiche (2001). Theorem 9 in the paper says that replacing non-priority literals with true literals in a d-DNNF is a proper projection operation, and Lemma 3 establishes logical equivalence between $D$ and $D_P$ modulo variables in $\mathcal{P}$.
10.2 Projected Model Counting

d2c(\text{Nodes})
\begin{align*}
\text{init}() \\
\text{for } (n \in \text{Nodes}) \\
\quad \text{if } (n \text{ is a literal } l) \\
\qquad \text{if } (\text{var}(l) \in \mathcal{N}) \\
\qquad \quad \text{litAtNode}[n] := \text{true} \\
\qquad \text{else } \text{litAtNode}[n] := l \\
\text{elif } (n = \text{op}(c_1, \ldots, c_k)) \\
\qquad (l_1, \ldots, l_j) := \text{simplify}(n) \\
\qquad \text{if } (j = 1) \\
\qquad \quad \text{litAtNode}[n] := l_1 \\
\text{else} \\
\qquad h := \text{hash}(\text{op}, (l_1, \ldots, l_j)) \\
\qquad \text{if } (\text{litWithHash}.\text{hasKey}(h)) \\
\qquad \quad \text{litAtNode}[n] := \text{litWithHash}[h] \\
\text{else} \\
\qquad v := \text{Tseitin}(\text{op}, (l_1, \ldots, l_j)) \\
\qquad \text{litAtNode}[n] := v \\
\qquad \text{litWithHash}[h] := v \\
\text{C.add}([\text{litAtNode}[\text{Nodes}.\text{last()}]]) \\
\text{return } C
\end{align*}

\begin{align*}
\text{init}() \\
\quad C = (), \text{litAtNode} = \{\}, \text{litWithHash} = \{} \\
\quad v := |\mathcal{V}|
\end{align*}

\text{simplify}(\text{op}(c_1, \ldots, c_k))
\begin{align*}
L = () \\
\text{for } (c \in c_1, \ldots, c_k) \\
\quad \text{if } (\text{litAtNode}[c] = \text{true}) \\
\qquad \text{if } (\text{op} = \text{OR}) \text{ return } \text{(true)} \\
\text{else} \\
\qquad L.\text{add}(\text{litAtNode}[c]) \\
\text{return } L
\end{align*}
Example 10.3. Consider the formula $F$ with priority variables $p, q$ and non-priority variables $x, y, z$:

$$(\neg x \lor p) \land (q \lor \neg x \lor y) \land (\neg p \lor \neg y \lor \neg z \lor q) \land (x \lor q) \land (\neg q \lor p)$$

The projected model count is 2 ($\{p, q\}$ and $\{p, \neg q\}$).

Figure 10.5: Example of application of d2c

Figure 10.5 shows the initial d-DNNF (10.5a), the DNNF obtained by replacing all non-priority literals by true and simplifying (10.5b) and the d2c translation of the projected DNNF (10.5c). Notice that if we perform model counting naively on the projected DNNF, we get a count of 3 since we double count the model $\{p, q\}$. The satisfaction probability is:

$$\left(\frac{1}{2} + \frac{1}{2}\right) \times \frac{1}{2} + \left(\frac{1}{2} \times \frac{1}{2}\right) = \frac{3}{4}$$

From satisfaction probability, we get the model count $2^2 \times \frac{3}{4} = 3$. However, if we count the models of the translated formula in 10.5c, we get the correct count of 2.

10.3 Experiments

We compare the following solvers on various benchmarks: CLASP in its projection mode (CLASP), our extension of clasp with cube minimization (#CLASP), model counting with searching on priority variables first (DSHARP, P), and counting models of projected DNNF (D2C). In each row $|P|$ is the number of priority variables. $T$ and $D$ represent the execution time and number of decisions taken by the solver. $R$ is a parameter to gauge the
quality of cubes computed by \#CLASP, the higher it is, the better. It is equal to \( \log_2 \left( \frac{\#sols}{\#cubes} \right) \).

A value of 0 indicates that all solution cubes computed have size 1, while the maximum value is equal to the number of priority variables, which is the unique case when there is only one cube and every assignment to priority variables is a solution. \( R \) essentially quantifies the advantage over enumeration, the less constrained a problem is, and the more general the cubes are, the higher the advantage. \( S \) is the size (in bytes) of the CNF computed by \( D2C \) that is subsequently given to the solver SHARPSAT for model counting.

The timeout for all experiments is 10 minutes. All times are shown in seconds. The experiments were run on NICTA’s HPC cluster that uses Rocks 6.1.1. and has AMD 6-Core Opteron 4184 CPUs.

### 10.3.1 Uniform Random 3-SAT and Boolean Circuits

Table 10.1 shows the results from uniform random 3-SAT and random Boolean circuits. In this table, for each problem instance, we show how the solvers perform as we increase the number of priority variables. A … after a row means that every solver either ran out of time or memory for all subsequent number of priority variables until the next one shown. For each instance, a row is added that provides the following information about it: name, number of solutions as reported by \( DSHARP \), number of variables and clauses and time and decisions taken by \( DSHARP \). Note that this time should be added to the time of \( D2C \) in order to get the actual time of \( D2C \) approach.

Let us look at the results from uniform random 3-SAT. All instances have 100 variables, and the number of clauses is varied. We try clause-to-variable ratios of 1, 1.5, 2, 3 and 4. Note that for model counting, the difficulty peaks at the ratio of approximately 1.5 (Gomes et al., 2009). For the first 3 instances, \#CLASP is the clear winner while CLASP also does well, DSHARP,P lags behind both, and D2C does not even work since the original instance cannot be solved by DSHARP. For \#CLASP, as we increase the number of clauses, the cube quality decreases due to the problem becoming more constrained and cube minimization becoming less effective. For 300 clauses, we see a significant factor coming into

---

28 All benchmarks and solvers are available at: http://people.eng.unimelb.edu.au/pstuckey/countexists
play for $\text{dSHARP}_P$. The original instance is solved by $\text{Dsharp}$. As we increase the number of priority variables until nearly the middle, the performance of $\text{dSHARP}_P$ degrades, but after 50 priority variables, it starts getting better. This is because the degradation due to searching on priority variables first becomes less significant and the search starts working more naturally in its VSADS mode (Sang et al., 2005). $\text{d2c}$ also solves two rows in this instance but is still largely crippled compared to other solvers. Finally, with 400 clauses, we are well past the peak difficulty and the number of models is small enough to be enumerated efficiently by $\text{CLASP}$. All solvers finish all rows of this instance in less than .15 seconds. We tried the same ratios for 200 and 300 variables. For 200 variables, we saw the same trend, although the problem overall becomes harder and the number of solved rows decreases. For 300 variables, the problem becomes significantly harder to be considered a suitable benchmark.

The Boolean circuits are generated with $n$ variables as follows: we keep a set initialized with the $n$ original variables, then as long as the set is not a singleton, we randomly pick an operator $o$ (AND, OR, NOT), remove random operands $V$ from the set, create a new variable $v$ and post the constraints $v \leftrightarrow o(V)$ and put $v$ back in the set. The process is repeated $c$ times. In the table, we show the results where $n$ is 30, and $c$ is 1,5,10. Note that a higher value of $c$ means that the problem is more constrained. Overall, for all instances, $\text{dSHARP}_P$ is the superior approach, followed by $\text{CLASP}$; and $\text{d2c}$ is better than $\#\text{CLASP}$ in $c = 1$ but the converse is true for higher values of $c$. All solvers find $c = 5$ to be the most difficult instance. We saw similar trends for different values of $n$ that have appropriate hardness with same values of $c$.

| $|P|$ | CLASP | $\#\text{CLASP}$ | $\text{dSHARP}_P$ | $\text{D2C}$ |
|-----|-------|--------------|-----------------|-----------|
|     | $|$    | $T$ | $D$ | $T$ | $D$ | $R$ | $T$ | $D$ | $S$ |
|-----|-------|-----|-----|-----|-----|-----|-----|-----|-----|
| 5   | 32    | .0  | 2271| 0   | 291 | 3.00| .04 | 1150| —   | —   | —   |
| 10  | 1024  | .01 | 71309| 0   | 533 | 7.19| .99 | 35927| —   | —   | —   |
| 15  | 32768 | .40 | 2023146| 0   | 1888| 10.30| 7.92| 370034| —   | —   | —   |
### 10.3 Experiments

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>2.7e+07</td>
<td>345.57</td>
<td>1.4e+09</td>
<td>0</td>
<td>10584</td>
<td>17.36</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>35</td>
<td>1.8e+10</td>
<td>—</td>
<td>—</td>
<td>.02</td>
<td>62016</td>
<td>24.32</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>50</td>
<td>1.9e+14</td>
<td>—</td>
<td>—</td>
<td>107.75</td>
<td>7.1e+07</td>
<td>27.04</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

...  

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>2.1e+07</td>
<td>270.91</td>
<td>8.7e+08</td>
<td>.34</td>
<td>393130</td>
<td>11.25</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>35</td>
<td>2.8e+09</td>
<td>—</td>
<td>—</td>
<td>24.98</td>
<td>1.5e+07</td>
<td>12.84</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

...  

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>2.1e+07</td>
<td>270.91</td>
<td>8.7e+08</td>
<td>.34</td>
<td>393130</td>
<td>11.25</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>35</td>
<td>2.8e+09</td>
<td>—</td>
<td>—</td>
<td>24.98</td>
<td>1.5e+07</td>
<td>12.84</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

...  

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>32</td>
<td>0</td>
<td>1937</td>
<td>0</td>
<td>247</td>
<td>3.00</td>
<td>.03</td>
<td>1286</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
<td>.02</td>
<td>54077</td>
<td>.01</td>
<td>2933</td>
<td>4.19</td>
<td>.68</td>
<td>27142</td>
</tr>
<tr>
<td>15</td>
<td>32768</td>
<td>.42</td>
<td>1767073</td>
<td>0</td>
<td>2101</td>
<td>9.96</td>
<td>31.63</td>
<td>1057551</td>
</tr>
<tr>
<td>25</td>
<td>2.1e+07</td>
<td>270.91</td>
<td>8.7e+08</td>
<td>.34</td>
<td>393130</td>
<td>11.25</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>35</td>
<td>2.8e+09</td>
<td>—</td>
<td>—</td>
<td>24.98</td>
<td>1.5e+07</td>
<td>12.84</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

...  

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>32</td>
<td>0</td>
<td>1354</td>
<td>0</td>
<td>259</td>
<td>2.42</td>
<td>.04</td>
<td>1304</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
<td>.01</td>
<td>47771</td>
<td>0</td>
<td>1370</td>
<td>5.25</td>
<td>1.12</td>
<td>37596</td>
</tr>
<tr>
<td>15</td>
<td>30712</td>
<td>.43</td>
<td>1408296</td>
<td>0</td>
<td>4659</td>
<td>8.29</td>
<td>37.07</td>
<td>874826</td>
</tr>
<tr>
<td>25</td>
<td>1.8e+07</td>
<td>218.20</td>
<td>6.4e+08</td>
<td>1.69</td>
<td>1801261</td>
<td>8.54</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

...  

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>32</td>
<td>0</td>
<td>986</td>
<td>0</td>
<td>646</td>
<td>0.75</td>
<td>.05</td>
<td>671</td>
</tr>
<tr>
<td>10</td>
<td>970</td>
<td>.02</td>
<td>25441</td>
<td>.02</td>
<td>11450</td>
<td>1.20</td>
<td>2.07</td>
<td>14146</td>
</tr>
<tr>
<td>15</td>
<td>12990</td>
<td>.22</td>
<td>290973</td>
<td>.11</td>
<td>61663</td>
<td>2.30</td>
<td>12.74</td>
<td>144211</td>
</tr>
<tr>
<td>25</td>
<td>226117</td>
<td>3.84</td>
<td>3432170</td>
<td>1.66</td>
<td>464908</td>
<td>3.21</td>
<td>57.57</td>
<td>808670</td>
</tr>
<tr>
<td>35</td>
<td>5126190</td>
<td>49.02</td>
<td>6.6e+08</td>
<td>15.65</td>
<td>3367386</td>
<td>4.67</td>
<td>161.67</td>
<td>2834211</td>
</tr>
<tr>
<td>50</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>61M</td>
</tr>
<tr>
<td>65</td>
<td>1.6e+09</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>70.80</td>
<td>1552565</td>
<td>—</td>
</tr>
</tbody>
</table>
### Projected Stable Model Counting

| UF #=45868 | V|=100 | | C|=160 | T=, .05 | D=244 |
|------------|------|---|---|---|---|
| n=30 c=1 | #=9.657e+08 | | V|=99 | | C|=167 | T=0 | D=111 |
| 5 | 16 | 0 | 409 | 0 | 292 | 0.54 | 0 | 113 | .01 | 4 | 1.1K |
| 9 | 160 | 0 | 3418 | 0 | 1184 | 1.54 | 0 | 143 | 0 | 8 | 1.2K |
| 14 | 552 | 0 | 9305 | 0 | 5030 | 1.00 | 0 | 82 | .01 | 22 | 3.8K |
| 24 | 248960 | 1.16 | 2718019 | 1.49 | 833682 | 2.07 | 0 | 130 | .01 | 169 | 6.1K |
| 34 | 1621760 | 6.13 | 1.2e+07 | 6.45 | 199088 | 3.13 | .01 | 111 | .05 | 656 | 9.7K |
| 49 | 3.9e+07 | 104.26 | 1.9e+07 | 353.25 | 1.4e+07 | 4.78 | .01 | 162 | .10 | 1393 | 14K |
| 64 | 1.5e+08 | 394.21 | 4.6e+08 | — | — | — | .01 | 108 | .21 | 2982 | 20K |
| 74 | 4.4e+08 | — | — | — | — | — | 0 | 99 | .20 | 2624 | 24K |
| 84 | 7.2e+08 | — | — | — | — | — | .01 | 111 | .20 | 2517 | 27K |
| 99 | 9.7e+08 | — | — | — | — | — | .01 | 111 | .20 | 2517 | 27K |
### 10.3 Experiments

| $n=30$ | $c=5$ | $\#=9.426e+07$ | $|V|=389$ | $|C|=867$ | $T=288.45$ | $D=1036363$ |
|-------|-------|-----------------|---------|---------|----------|-------------|
| 19    | 12192 | .16             | .75     | 146058  | 0.00     | 16.48      | 120619      | —         | 5.8M      |
| 38    | 208716| 2.57            | 1882991 | 15.23   | 1985136  | 0.00       | 95.37      | 834705     | —         | 47M       |
| 58    | 1.2e+07| 93.69          | 3.7e+07 | —       | —        | —          | —          | —         | 100M      |
| 97    | 3.3e+07| 248.76         | 6.9e+07 | —       | —        | —          | 427.85     | 1509308   | —         | 171M      |
| 136   | 6.1e+07| 428.89         | 9.1e+07 | —       | —        | —          | —          | —         | 252M      |
| ...   |       |                |         |         |   | | | | |
| 291   | 9.3e+07| —              | —       | —       | —        | —          | 300.78     | 985065    | —         | 574M      |
| 330   | 9.4e+07| —              | —       | —       | —        | —          | 299.02     | 1074927   | —         | 672M      |
| 389   | 9.4e+07| —              | —       | —       | —        | —          | 308.84     | 1036363   | —         | 783M      |

| $n=30$ | $c=10$ | $\#=5066$ | $|V|=766$ | $|C|=1771$ | $T=.32$ | $D=1400$ |
|-------|-------|-----------|-----------|-----------|--------|---------|
| 38    | 282   | .01       | 1196      | .03       | 1797   | 0.00    | .31     | 1412     | .08     | 256     | 36K      |
| 76    | 1618  | .02       | 3479      | .12       | 5434   | 0.00    | .40     | 1600     | .81     | 2046    | 137K     |
| 114   | 2581  | .03       | 4984      | .21       | 7953   | 0.00    | .52     | 1787     | 1.69    | 3702    | 173K     |
| 191   | 4948  | .05       | 5558      | .52       | 12243  | 0.00    | .54     | 1904     | 4.98    | 7519    | 330K     |
| 268   | 5066  | .07       | 5458      | .63       | 12235  | 0.00    | .38     | 1784     | 6.69    | 10508   | 478K     |
| 383   | 5066  | .09       | 5513      | .85       | 12356  | 0.00    | .69     | 1975     | 9.27    | 12528   | 698K     |
| 497   | 5066  | .08       | 5253      | 1.47      | 12471  | 0.00    | .39     | 1680     | 12.46   | 11818   | 1.1M     |
| 574   | 5066  | .11       | 5211      | 1.68      | 12358  | 0.00    | .52     | 1616     | 14.85   | 11911   | 1.1M     |
| 651   | 5066  | .09       | 5500      | 1.21      | 12546  | 0.00    | .36     | 1500     | 13.42   | 11807   | 1.4M     |
| 766   | 5066  | .09       | 5072      | 2.06      | 12389  | 0.00    | .33     | 1400     | 21.61   | 11644   | 1.6M     |

Table 10.1: Results from random uniform 3-SAT and Boolean circuits
10.3.2 Planning

Table 10.2 summarizes performance of different projected model counting algorithms on checking robustness of partially ordered plans to initial conditions. We take five planning benchmarks: depots, driver, rovers, logistics, and storage. For each benchmark, we have two variants, one with the goal state fixed and one where the goal is relaxed to be any viable goal (shown with a capital A in the table representing any goal). For the two variants, the priority variables are defined such that by doing projected model counting, we count the following. For the first problem, we count the number of initial states the given plan can achieve the given goal from. For the second problem, we count the number of initial states plus all goal configurations that the given plan works for. Each row in the table represents the summary of 10 instances. The first 3 columns show the instance parameters. For each solver, ✓ shows how many instance the solver was able to finish within time and memory limits. All other solver parameters are averages over finished instances. Another difference from the previous table is that we have added the execution time of DSHARP in D2C and DSHARP time is shown in parenthesis. There was no case in which only DSHARP finished and the remaining steps of D2C did not finish.

Overall, DSHARP solves the most instances (42), followed by #CLASP (41), D2C (34), and finally CLASP which solves only 4 instances from the storage benchmark, and otherwise suffers due to the inability to detect cubes. DSHARP and D2C only fail on all instances in 2 benchmarks while #CLASP fails in 4, so they are more robust in that sense. For D2C, the running time is largely taken by producing the d-DNNF and the second round of model counting is relatively cheaper. The cube quality of #CLASP is quite significant for all instances that it solves.

10.4 Projected Stable Model Counting

In this section, we describe how to extend the projected model counting algorithms described previously, to projected stable model counting. For this section, we are given an ASP program \( P = (V, R, C) \), the priority variables \( P \subseteq V \). The projected stable model count is defined as the number of complete assignments \( \theta_P \) to variables in \( P \) such
10.4 Projected Stable Model Counting

that there exists an assignment \( \theta_N \) to the non-priority variables \( N = V \setminus P \), such that \( \theta = \theta_P \cup \theta_N \) is a stable model of \( P \), i.e., \( \theta \) satisfies all constraints in \( C \) and \( \text{Least}(R^\theta) = V^\theta \).

10.4.1 \texttt{dSHARP\_P}

\texttt{dSHARP\_P} can be extended from the algorithm presented in 10.2.1 to stable model counting in a straight-forward way. We can do so by adding unfounded set detection in the model counting algorithm and considering the justified residual program instead of the residual program for the purposes of cube detection, caching, and dynamic decomposition as we described in the previous chapter. The projection part of the algorithm, which includes projecting the count of a component over variables in \( P \) and backjumping to the last component that contains priority variables, remains the same as described in 10.2.1.

10.4.2 \texttt{#CLASP}

The algorithm that we defined for projected model counting in 10.2.2 extends the solution enumeration algorithm of Gebser et al. (2009b), which works for both ASP and SAT. We now consider how we can shrink the cube of a logic program that has been found by a conflict-driven ASP solver, such as \texttt{CLASP}. Below we provide the procedure \texttt{shrinkASP} that can be used in place of the previous \texttt{shrink} procedure described in Subsection 10.2.2.
Recall that the goal of shrink procedure in that subsection is to minimize a solution cube for a CNF $C$, in presence of blocking clauses $B$.

\begin{verbatim}
shrinkASP(\theta)
S := \theta_N \cup \theta_Y \cup \{dec(i) : i \in 1 \ldots bl\} \quad \% \text{initializing } S

\textbf{while } (R|_S^j \neq \emptyset \text{ and } (C \cup B)|_S \neq \emptyset)
Q = \{l_1, \ldots, l_k : h \leftarrow l_1, \ldots, l_k \in R|_S^j : \{l_1, \ldots, l_k\} \subseteq \theta_Y\}
\cup \{p \in cl, cl \in (C \cup B)|_S\}
p = Q.pop()
S.add(p)
S := S \setminus p
\text{count := count } + 2^{|P| - |S|}
B.add(\neg S)
\end{verbatim}

Figure 10.6: Pseudo-code for shrinking a solution $\theta$ of an original ASP program $P = (V, R, C)$ and blocking clauses $B$ to a solution cube $S$, adding its count and a blocking clause to prevent its reoccurrence.

shrinkASP minimizes a solution cube $\theta$ for an ASP program $P = (V, R, C)$, when we have existing blocking clauses $B$ (initially empty). The pseudo-code is similar to shrink, so we only comment on the way this algorithm differs from it. First, note that a founded variable can never be unfixed in a solution cube. If it is unfixed, then it must have at least one rule in the residual, otherwise, it should be false according to the stable model semantics. Since the residual is not empty, the current assignment cannot be a cube. Therefore, to simplify the procedure, we add all founded variables in $S$ in the initialization. Then we incrementally build up $S$ until the residual of clauses (including blocking) w.r.t. $S$ is empty and there are no unsupported bodies hanging in the justified residual rules $R|_S^j$. Recall that $R|_S^j$ is the set of residual rules w.r.t. the justified assignment $JA(P, \theta)$. A non-empty set indicates that there is at least one standard literal in $\theta$ appearing in the body of unjustified true founded variable that needs to be added to $S$ to justify it. After the while
loop, the rest of the pseudo-code is the same as in shrink.

10.4.3 D2C

It is straight-forward to extend the approach described in 10.2.3 to do projected stable model counting. The only change applies in the first part, where, instead of the d-DNNF of a CNF, we build a d-DNNF that is logically equivalent to the set of stable models of the logic program. We have already described a stable model knowledge compiler in the previous chapter. Projecting away the literals $N$ from the d-DNNF produced by the stable model compiler and then counting the models of Tseitin encoding of that d-DNNF is done in the same way as the one described for CNF.

10.5 Conclusion

The area of Boolean Quantifier Elimination (BQE) seems closely related to projected model counting. Although the goal in BQE is to produce a non-priority variables free representation (usually CNF), some algorithms can also be adapted for projected model counting. Of particular interest are techniques in (Brauer et al., 2011) and (Goldberg and Manolios, 2012). The algorithm of (Brauer et al., 2011) finds cubes in decreasing (increasing) order of cube size (number of literals in cube). While this approach does not require cube minimization, it does not run in polynomial space, and if a problem only has large cubes, then significant time might be wasted searching for smaller ones. The second interesting approach, with promising results, is given in (Goldberg and Manolios, 2012), which uses a DPLL-style search and also decomposes the program at each step like the approach described in 10.2.1. A good direction for future work is to investigate how well these techniques lend themselves to projected model counting and whether there is any room for integration with the ideas presented in this chapter.

In this chapter we compare four algorithms for projected model counting. We see that each algorithm can be superior in appropriate circumstances:

- When the number of solutions is small then CLASP (Gebser et al., 2009b) is usually the best.
• When the number of solution cubes is much smaller than solutions, and there is not much scope for component caching, then \#CLASP is the best.

• When component caching and dynamic decomposition are useful then DSHARP_P is the best.

• Although d2c is competitive, it rarely outperforms both \#CLASP and DSHARP_P.

Having said that, d2c approach has another important aspect besides projected model counting. It is a method to perform projection on a d-DNNF without losing determinism. This can be done by computing the d-DNNF of the CNF produced by the d2c procedure (instead of model counting), and then simply forgetting the Tseitin variables (replacing with true). It can be shown that this operation preserves determinism. Furthermore, our experiments show that the last model counting step takes comparable time to computing the first d-DNNF in most cases (and in many cases, takes significantly less time), which means that the approach is an efficient way of performing projection on a d-DNNF.

While we use d-DNNF for d2c approach, it is possible to use other, less succinct, languages like Ordered Binary Decision Diagrams (OBDDs). We leave the comparison with other possible knowledge compilation-based approaches for projected model counting as future work. As the problem of projected model counting is not heavily explored there is significant scope for improving algorithms for it. A simple improvement would be to portfolio approach to solving the problem, combining all four of the algorithms, to get the something close to the best of each of them.
Part III

Conclusion
Chapter 11

Conclusion

This thesis contributes to theory and practice of Answer Set Programming. In this chapter, we present the summary of our contributions.

We describe a Constraint ASP implementation within the lazy clause generation solver CHUFFED. This approach is unique in the sense that previous implementations for CASP either combine an ASP solver with a CP solver, or are translation based. Our implementation is competitive with state-of-the-art CASP solver CLINGCON.

We present Bound Founded ASP, a powerful and expressive formalism that generalizes ASP, CP, CASP, and Fuzzy ASP. BFASP allows us to model foundedness over numeric variables and supports a rich set of rule forms, as opposed to a few rigidly defined rule forms used in ASP. We demonstrate the use of BFASP by showing how problems like road construction, utilitarian policies, and company controls (Chapter 6) can be elegantly modelled in BFASP. To implement BFASP, we first extend the definition of unfounded sets for BFASP, and then extend the source pointer algorithm which is a well-known technique for implementing Boolean unfounded sets, to work for founded numeric variables and more general rule forms in BFASP. Our implementation of this algorithm in CHUFFED shows a remarkable speed-up on the BFASP benchmarks as compared to ASP models.

We look at the language aspects of BFASP. We extend the language technique called flattening, which is defined for constraint languages and transforms arbitrary (type-correct) expressions to a primitive set of constraints, to BFASP. We then borrow two very useful grounding algorithms from logic programming, semi-naive evaluation or bottom-up grounding and magic set transformation, and generalize them for BFASP. Our implementation of both these techniques show promising results on structured benchmark
instances.

We describe an algorithm for counting the stable assignments of a logic program and apply it to probabilistic logic programming. We take three techniques developed in propositional model counting, namely cube detection, caching, and dynamic decomposition, and show how these can be extended theoretically for logic programs under stable model semantics. The core contribution lies in identifying the justified residual logic program with respect to a partial assignment, and doing so enables a convenient extension of the aforementioned techniques in model counting to work for stable model counting. We describe two ways to compute justified residual programs in a propositional model counter. The first approach only modifies the search of the model counter such that the propositional residual is equal to the justified residual. The second approach is a more complete approach that introduces more variables and clauses to model the justified residual program, and does not modify the search heuristic on the original variables of the logic program. We implement our stable model counting algorithm in the probabilistic logic programming tool-chain PROBLOG2 and show that for certain problems, it outperforms the state-of-the-art implementation that relies on translating a logic program under stable model semantics to a propositional theory.

Finally, we look at the problem of projected stable model counting, in which the goal is to count the stable assignments of a logic program projected on a subset of the original set of variables. We approach this by first looking at projected propositional model counting, and then describe how those approaches can be extended for projected stable model counting. We look at three approaches for projected model counting, two of which are novel, and compare them on different benchmarks. Our results show that all three approaches can be useful depending on the problem instance and there is no conclusive winner. Our implementation and experiments are limited to projected model counting, and extending them for projected stable model counting is a good candidate for future work.
Bibliography
Bibliography


Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. DPLL with Caching: A new algorithm for #SAT and Bayesian Inference. Electronic Colloquium on Computational Complexity (ECCC), 10(003), 2003.


Marcello Balduccini. Representing constraint satisfaction problems in answer set programming. In ICLP09 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP09), 2009.


Daan Fierens, Guy Van den Broeck, Ingo Thon, Bernd Gutmann, and Luc De Raedt. Inference in probabilistic logic programs using weighted CNF’s. In Fabio Gagliardi Cozman


Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal


Minerva Access is the Institutional Repository of The University of Melbourne

Author/s:
Aziz, Rehan Abdul

Title:
Answer set programming: founded bounds and model counting

Date:
2015

Persistent Link:
http://hdl.handle.net/11343/58612

File Description:
Answer Set Programming: Founded Bounds and Model Counting