FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

OMiGA

An Open Minded Grounding on-the-fly Answer Set Solver

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Gerald Weidinger
Matrikelnummer 0526105

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Prof. Dr. Thomas Eiter
Mitwirkung: Dr. Michael Fink
Dipl. Inf. Antonius Weinzierl

Wien, 25.08.2013

(Unterschrift Gerald Weidinger) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

OMiGA

An Open Minded Grounding on-the-fly Answer Set Solver

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Gerald Weidinger

Registration Number 0526105

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Prof. Dr. Thomas Eiter
Assistance: Dr. Michael Fink
Dipl. Inf. Antonius Weinzierl

Vienna, 25.08.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Gerald Weidinger
Franz Liszt- Gasse 24, Mattersburg 7210

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Gerald Weidinger)

Acknowledgements

I want to thank Dipl. Inf. Antonius Weinzierl for supporting me over the whole process of this thesis. For giving me advices on background and solver architecture, for interesting conversation on the topic of Logic, ASP and Multi-Context-Systems, for implementing the OMiGA parser component and for bearing with me this long to help me formulate this thesis in a scientific way.

I want to thank Dr. Michael Fink for bringing me to the topic of ASP-Solving and Multi-Context-Systems and for giving me the chance to write this thesis, as well as for the support given during writing this thesis, especially for help on the proof of the theoretical algorithm.

I want to thank Prof. Dr. Thomas Eiter for several advices on the logical background of this thesis as well as allowing me to write this thesis and for correcting and improving it up to this point.

Finally I want to thank my girlfriend and my family for supporting me during my work on this thesis and for believing in me during my academic studies.

Abstract

Answer Set Programming (ASP) is nowadays a well known and well acknowledged declarative problem solving paradigm. ASP originates from the area of artificial intelligence and is often used in knowledge-based reasoning and logic programming, where its main purpose is solving search and optimization problems. Answer Set Programming is directly rooted in first-order logic and its programs are interpreted by the stable-model semantics introduced by Gelfond and Lifschitz. Several solvers have been implemented over the last years that are capable of evaluating such answer set programs. The standard approach to do this is usually separated into two steps, the grounding phase and the solving phase. In the grounding phase the input ASP program is parsed and a pre-grounding is created which contains for each nonground rule all those ground rule instances that might be derived later. In the second step those grounded rules are applied when possible and guesses are done when needed to solve the input ASP program and return all its answer sets.

While this approach is easy to understand and offers opportunities for many optimizations since the solver only needs to deal with ground rules, one problem arises: an exponential blowup. In general, when grounding an ASP program, the grounding can become exponential which, in the worst case needs more space than available. On the other side there are solvers using grounding on-the-fly, a technique that enables a solver to skip the pre-grounding phase and immediately start solving. The benefit of this approach is that rules are grounded while solving, so only if they are really needed to further calculate an answer set. A solver using grounding on-the-fly can create smaller groundings than standard solvers because during its solving phase, a solver can take into account which atoms have already been derived and which must not be derived anymore, therefore reducing the number of possible rules to ground tremendously. While of course in general an exponential blowup of the grounding can not be avoided since it can be part of the input program's nature, the grounding on-the-fly approach indeed creates smaller groundings than the standard one, and is capable of solving some program instances where standard solvers yield because of insufficient memory.

But from grounding on-the-fly a new challenge arises: Finding those ground rule instances for a nonground rule that are applicable. Since grounding on-the-fly is a rather new approach there are not many solvers implementing this technique, and all of those known to us have troubles dealing with this problem.

In this thesis we present OMiGA, a modern ASP solver implemented in Java. OMiGA efficiently uses grounding on-the-fly, while offering structured methods to manage the calculation of answer sets from outside. This enables the programmer to use OMiGA easily within a Multi-Context System, since propagation, choice and backtrack can be simply triggered from outside in order to compute answer sets iteratively. To address the problem of exponential blowup the OMiGA solver combines the grounding on-the-fly approach with a Rete network. A Rete network is a graph-like structure that is known from rule based production and expert systems. The benefit of a Rete network is that partial rule interpretations are stored within the nodes of the network and therefore need not be recalculated at each calculation step like in a naive approach. This can increase the performance tremendously.

Within this thesis we present the theoretical algorithm behind OMiGA including a proof of correctness as well as a runtime analysis. We furthermore introduce the reader to the OMiGA system architecture and OMiGA's own Rete network which has been optimized for use with ASP rules. Finally we present an evaluation over several benchmark instances to compare OMiGA with several other ASP solvers to show strengths and weaknesses of our implementation.

Kurzfassung

Answer Set Programming (ASP) ist ein deklaratives Paradigma zum Lösen von Problemen, dessen Ursprung auf das wissenschaftliche Gebiet der künstlichen Intelligenz zurückzuführen ist und hauptsächlich zum Lösen von Such- und Optimierungs-Problemen in den Bereichen Wissensbasiertes Schließen und Logikorientierte Programmierung eingesetzt wird. Answer Set Programming basiert auf der Prädikatenlogik erster Stufe und Programme die in ASP geschrieben sind werden mit Hilfe der der Semantik der stabilen Modelle (Gelfond und Lifschitz) interpretiert. Im Laufe der letzten Jahre wurden einige Solver geschrieben die sich auf das Evaluieren solcher Programme spezialisiert haben. Der herkömmliche Ansatz beruht hierbei auf einem zweistufigen System bei dem zuerst ein Pre-Grounding für das Inputprogramm erstellt wird, welches dann im zweiten Schritt verwendet wird um Answersets zu berechnen. Das Pregrounding besteht hierbei aus allen grundierten Regelinstanzen die während der Berechnung in Phase zwei benötigt werden könnten. In der Berechnungsphase werden diese Regelinstanzen, falls sie anwendbar sind, angewandt und falls nötig diverse Regeln als wahr angenommen oder verworfen um alle Answersets des Inputprogramms zu berechnen.

Dieser Ansatz ist grundsätzlich leicht zu implementieren und bietet viele Möglichkeiten zur Optimierung, da in der Berechnungsphase nur noch mit grundierten Regelinstanzen zu arbeiten ist. Aus dem zweistufigen Konzept ergibt sich jedoch ein großer Nachteil: der exponentielle Speicherverbrauch des Pregroundings. Generell kann das Grounding eines ASP Programms exponentiell groß werden und somit im schlimmsten Fall so viel Speicher benötigen, dass dieses nicht mehr im Speicher eines Computers Platz findet.

Aus diesem Grund gibt es seit kurzer Zeit ASP Solver die dynamisches Grounding betreiben. Im Gegensatz zum vorher erläuterten zwei Stufen Ansatz werden die Grundinstanzen der Regeln des Inputprogramms während der Berechnungsphase nach Bedarf erzeugt. Dies birgt den Vorteil, dass der Solver zu jedem Zeitpunkt genau weiß welche Fakten gelten und somit auch welche Regeln im nächsten Berechnungsschritt eine Rolle spielen können beziehungsweise welche Regeln sicher nicht mehr zur Anwendung kommen. Ein exponentielles Grounding ist im Allgemeinen zwar nie vermeidbar, da das Input Problem eine exponentielle Komplexität haben kann, jedoch sind diese dynamischen Solver in der Lage weit kleinere Groundings zu erstellen, als ihre zweistufigen Gegenstücke.

Das dynamische Grounding führt jedoch zu der Herausforderung feststellen zu müssen, welche Grundinstanzen im nächsten Berechnungsschritt angewandt werden können. Da das dynamische Grounden ein eher modernerer Ansatz ist, gibt es noch nicht viele Implementierungen die diesen umsetzen, und alle uns bekannten weisen hier Probleme auf. Der OMiGA-Solver soll dies nun mit Hilfe eines Rete-Netzwerks lösen. Ein Rete-Netzwerk enthält Knoten die Teile von Regeln darstellen und die zugehörigen Grundinstanzen dieser Teilregeln speichern. Der Vorteil liegt darin, dass durch die Speicherung von Teilregeln diese nicht jedes Mal neu berechnet werden müssen, im Gegensatz zu einem naiven Ansatz. Dies führt zu einer extremen Performance Steigerung.

In dieser Arbeit wird der in Java implementierte Solver OMiGA vorgestellt. OMiGA besticht durch seine effiziente grounding on-the-fly Lösung, sowie durch seine strukturierten Methoden die es dem Programmierer ermöglichen die unterschiedlichen Schritte zum Berechnen von Answersets (Propagation, Choice, Backtrack) von außen zu steuern und so eine iterative Berechnung durchzuführen. Dies macht OMiGA besonders interessant für den Einsatz in Multi-Context Systems, wo Berechnungen über mehrere Wissensbasen verteilt durchgeführt werden müssen und demnach ein iterativer Berechnungsansatz notwendig ist um die unterschiedlichen Kontexte miteinander zu koordinieren.

In dieser Arbeit wird zuerst der theoretische Algorithmus hinter OMiGA präsentiert, inklusive eines Beweises für Korrektheit und eine Laufzeitanalyse. Danach geben wir tieferen Einblick in die Systemarchitektur von OMiGA und das eigens entwickelte Rete-Netzwerk welches extra für den Umgang mit ASP Regeln implementiert wurde. Schließlich präsentieren wir Benchmark Ergebnisse um OMiGA in Relation zu anderen Solvern zu evaluieren und Stärken sowie Schwächen der aktuellen Implementierung aufzuzeigen.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Answer Set Programming (ASP) [22] is nowadays a well known and well acknowledged declarative problem solving paradigm. ASP originates from the area of artificial intelligence and is often used in knowledge-based reasoning and logic programming, because of its high expressiveness and for its ability to deal with incomplete knowledge [1]. Its main purpose is solving search and optimization problems. Answer Set Programming is directly rooted in first-order logic and its programs are interpreted by the stable-model semantics introduced by Gelfond and Lifschitz [12]. Over the last years many solvers have been implemented and ASP has been extended by many features like for example strong negation or disjunction. In the following we introduce the reader to the basic concept of ASP, and the problem of exponential blowup arising from using the standard solving approach. Then we present our approach which has been implemented as part of this Thesis: The grounding on-the-fly answer set solver OMiGA.

In ASP a program is a set of rules. A rule, for example, can look like this: $fly(X) : - bird(X)$, $not\, penguin(X)$., stating that a bird flies unless it is a penguin. In order to specify knowledge there are rules called facts e.g. $bird(tweety)$., stating that tweety is a bird. When combining ASP rules one can create an ASP program that is capable of solving many relevant problems. The semantics of ASP programs is given in terms of answer sets, which are certain sets of atoms.

Example 1.0.1. *The following example states that tweety is a bird and nora is a penguin. Furthermore it contains the knowledge that every penguin is also a bird, and that if something is a bird but no penguin then it will fly.*

$$\begin{aligned} bird(tweety). \quad & penguin(nora). \\ fly(X) : & -bird(X), not\, penguin(X). \\ bird(X) : & -penguin(X). \end{aligned}$$

In this case there is exactly one answer set, containing the initial knowledge that tweety is a bird and nora is a penguin, as well as that nora is a bird and that tweety can fly i.e. $A = \{bird(tweety), bird(nora), penguin(nora), fly(tweety)\}$.

While ASP is a theoretical approach to deal with problems, there are actually several implementations to compute answer sets [8, 16, 18, 27] available, so called ASP solvers. The standard technique to solve an ASP program is a two phase approach. In the first phase the rules of the input program that contain variables are grounded. This is done by replacing variable occurrences by elements of the domain to obtain all possible combinations of ground rules. So for Example 1.0.1 the rule $fly(X) : -\, bird(X), not\, penguin(X)$. would be grounded to :

$$\mathit{fly}(tweety) : -bird(tweety), not\, penguin(tweety).$$
$$\mathit{fly}(nora) : -bird(nora), not\, penguin(nora).$$

Then in the second phase, the solving phase, the answer sets are derived from those grounded rules. There are several techniques to do this. For example one can try to apply rules to gain new knowledge until an answer set is reached. In case of contradiction one has to backtrack and try different rules. The most popular ASP solvers using such an two phase approach are clasp [8] and DLV [18].

But the creation of this grounding can use too much space. This is due to the exponential blowup that can happen when grounding an ASP program. In general, theoretical results show that in grounding the exponential blowup can not be avoided, but for many cases optimizations can be done to speed up the process and create smaller groundings, by simply not creating rules that are not of interest. Standard solvers have developed a technique called intelligent grounding [6, 10] to optimize this grounding process. But from the very structure of this approach a disadvantage arises. Since the grounding is done before the actual solving phase, in general there is not enough knowledge to avoid all unnecessary rules. One way to address this problem is grounding on the fly, a technique that dynamically creates the grounding while solving. Hereby only ground rules are created whose body is fulfilled at the current calculation step.

Solvers implementing this approach are ASPERIX [16] and GASP [27]. These solvers are able to solve certain problems where standard solvers run out of memory. But their implementation to solve non ground rules is not very efficient, therefore those solvers can not compete with standard solvers in terms of speed on instances where exponential blowup can be avoided. A major issue responsible for this loss of speed is the trouble arising from determining all applicable ground rules of a non ground rule with respect to the current interpretation.

In this thesis we present the ASP solver OMiGA. Our goal is to create an ASP solver that has the advantage of a grounding of the fly solver while being able to keep up with the speed of a standard solver. To achieve this we address the issue of determining applicable ground rules by using a dynamic grounding procedure that is realized via a Rete network [7]. We also present an algorithm for solving ASP in this stepwise manner and show that this algorithm is sound and complete. Finally we run benchmarks to evaluate OMiGA in comparison to other grounding on the fly solvers as well as to standard solvers to show strengths and weaknesses of OMiGA.

For the practical part of this thesis OMiGA was implemented with main focus on the development of the underlying Rete network. Our Rete network is a graph like structure where each node represents an atom or a partial rule body. Each node has its own memory, where the corresponding grounded partial rule interpretations are stored. Whenever a new fact is derived only those nodes are updated where this fact contributes to. This way non ground rules get grounded step by step and the partial grounded ones are kept in the memory for later use, giving OMiGA a huge advantage over other grounding on the fly solvers that need to recalculate all those instances each time.

Our evaluation shows that OMiGA offers a faster calculation than ASPERIX, which seems to be the fastest grounding on the fly solver at the moment. Furthermore OMiGA can keep up even with the calculation speed of standard solvers running problem instances where only a small number of guesses is needed. Our evaluation of the graph reachability problem shows that using a Rete network is a good approach, since for this problem OMiGA is even faster than DLV; although reachability can be solved by intelligent grounding alone without entering an actual solving phase. For another benchmark example, called cutedge, OMiGA finishes calculation 20 times faster than clasp and is 10 times faster than ASPERIX for small instances, while for huge instances only OMiGA returned a result in plausible time.

The main focus of the implementation of OMiGA was to create an efficient mechanism for grounding on the fly within an ASP solver. Because of this OMiGA lacks heuristics and has only limited optimization techniques in terms of guessing, learning and backtracking; these are issues left for future work. Therefore as one can see in the 3COL evaluation OMiGA is rather slow at problems that require a large number of guesses. Nevertheless OMiGA is three orders of magnitude faster than ASPERIX in 3COL and we have reached our primary goal.

The remainder of this work is structured as follows. Chapter 2 starts with an introduction to answer set programming, introducing notation and basic knowledge that is needed to describe OMiGA's algorithm. In Chapter 3 OMiGA's base functions are defined, the algorithm is described and a proof is given to show completeness and correctness. Furthermore a complexity analysis is given. Chapter 4 introduces the reader to the rete network and how to use it to represent an ASP program. Chapter 5 offers a detailed description on the system architecture of OMiGA including a description on all entities used, as well as on the Rete network and calculation units. Chapter 6 evaluates OMiGA in comparison to other solvers showing OMiGAs strengths and weaknesses. Finally in Chapter 7 related work is presented. Future work is discussed in the conclusion (Chapter 8).

CHAPTER 2

Preliminaries

In this chapter we introduce the reader to Answer Set Programming (ASP), to get a general understanding on how an ASP solver might work.

ASP is built upon a first order language of constants, variables, terms, predicates, atoms, and literals. P denotes the set of all predicates. For every $p \in P$, n_p denotes the arity of p. Let $\mathcal{C}$ (resp. $\mathcal{V}$) denote the set of possible constants (resp. variables), such that $\mathcal{C} \cap \mathcal{V} = \emptyset$. A term is either a *constant* or a *variable*. An atom is of this form: p($t_1, ..., t_{n_p}$) where $p \in P$ and $t_1, ..., t_{p_n}$ are terms. A literal is either of the form a or $not\, a$, where a is an atom. An **ASP program** P now is a finite set of rules of the form

$$a : -b_1, ..., b_l, not\, c_1, ..., not\, c_m.$$

where a, $b_1, ..., b_l$, and $c_1, ..., c_m$ are atoms and not is the sign for negation-as-failure (i.e. the following atom is assumed not to hold). Let r be a rule of this form; then the head a of r is denoted by $head(r) = \{a\}$. The set of positive atoms is denoted by $body^+(r) = \{b_1, ..., b_l\}$ while the set of negative atoms is denoted by $body^-(r) = \{c_1, ..., c_m\}$, and $body(r) = body^+(r)$ $\cup body^-(r)$.

We can distinguish between several types of rules. A rule r is

- a fact, iff $body(r) = \emptyset$. (when writing facts we omit „:-“);
- a constraint, iff $head(r) = \emptyset$;
- a positive rule, iff $body^-(r) = \emptyset$;
- a non-monotonic rule, iff $body^-(r) \neq \emptyset$;
- is ground if r does not contain any variable.

Definition 2.0.1. *A rule r is safe, iff each variable v that is contained in any atom of $head(r)$ $\cup body^-(r)$ is also contained in $body^+(r)$. A program P is safe if all rules of P are safe.*

Note that many ASP solvers depend on some safety restriction on rules in order to solve ASP programs. In order for a program P to be solvable by our solver OMiGA, all rules of P have to be safe.

Example 2.0.2. *Within this example we have two facts in* (2.3)*, meaning that tweety is a bird, and nora is a penguin. With the positive rule in* (2.1) *we state that every penguin is also a bird. And in* (2.2) *we have a non-monotonic rule stating that a bird usually flies, if it is not a penguin.*

$$bird(X) :- penguin(X). \tag{2.1}$$

$$fly(X) :- bird(X), not\, penguin(X). \tag{2.2}$$

$$bird(tweety). \quad penguin(nora). \tag{2.3}$$

2.1 Answer set semantics

Since we are dealing with ASP programs, the semantics is given in terms of answer sets, also called stable models. The stable models semantics was first introduced in [12]. The following section is based on this paper.

Answer sets are defined with respect to ground programs. In order to obtain a ground program from a nonground program a domain is necessary. In general the domain of P is infinite. In order to practically solve an ASP program we therefore work with the active domain C_P.

Definition 2.1.1. *For an ASP program P let C_P denote the set of all constants that occur in P.*

If there is no constant contained within P then $C_P = c$ for some $c \in \mathcal{C}$. If P is clear from the context we will omit it, and only write C for the active domain. In case of Example 2.0.2 $C_P = \{nora, tweety\}$.

Grounding

In order to evaluate an ASP program P, the rules of P have to be grounded. In order to do so we replace each rule that contains variables, by all its ground instances. A *ground instance* of a rule is obtained by substituting each variable by a constant from C. To obtain all ground instances of a rule we have to create all such combinations. For a rule r we obtain v^c ground instances, where v is the number of variables occurring within r and $c = |C|$. By doing this for each rule of P we obtain $gr(P)$, an ASP program without variables.

Definition 2.1.2. *For a program P, let $gr(P)$ denote the set of all ground rules that may result from P with respect to C.*

Note that $gr(P)$ is bounded by the number of rules, variables and constants of P.

Example 2.1.1. *For demonstration we continue with our previous example (2.0.2), and calculate $gr(P)$ for it.*

$$bird(tweety) :- penguin(tweety). \tag{2.4}$$
$$bird(nora) :- penguin(nora). \tag{2.5}$$
$$fly(tweety) :- bird(tweety), not\, penguin(tweety). \tag{2.6}$$
$$fly(nora) :- bird(nora), not\, penguin(nora). \tag{2.7}$$
$$bird(tweety). \quad penguin(nora). \tag{2.8}$$

Nothing changes with the facts as they were ground to begin with, but one can see that both rules have been replaced by four ground rules. This is due to the fact that we had two constants within C, and each rule contained only one variable. That is why we just had to replace X once by tweety and once by nora for each rule.

Example 2.1.2. *If there was a rule containing 2 different variables X and Y, this rule would have been grounded to 4 ground instances, like for example the rule $pair(X,Y) :- birdy(X)$, $bird(Y)$., which would be grounded like following, in respect to the $C = \{tweety, nora\}$:*

$$pair(tweety, nora) :- bird(tweety), bird(nora). \tag{2.9}$$
$$pair(tweety, tweety) :- bird(tweety), bird(tweety). \tag{2.10}$$
$$pair(nora, tweety) :- bird(nora), bird(tweety). \tag{2.11}$$
$$pair(nora, nora) :- bird(nora), bird(nora). \tag{2.12}$$

Interpreting an ASP program

In order to interpret an ASP program one has to determine which ground atoms hold and which do not. This is given in terms of an interpretation defined as follows:

Definition 2.1.3. *Let the Herbrand Base of P be defined as $\mathcal{AT} = \{p(t_1, ..., t_{p_n}\} \, |t_1...t_{p_n} \in C, p \in P\}$. Then an* interpretation *I of P is a pair of sets $\langle In, Out\rangle$, where $In \subseteq \mathcal{AT}, Out \subseteq \mathcal{AT}$.*

Intuitively In contains all facts that hold, and Out contains all facts for which we know they must not hold. The set of all such interpretations is denoted by $\mathcal{I}$. For an Interpretation $I = \langle In, Out\rangle$ we denote by $I.IN$ (resp. $I.OUT$) the set In (resp. Out). Furthermore we write $I \subseteq I'$ if $I.IN \subseteq I'.IN$ and $I.OUT \subseteq I'.OUT$, and $I \subset I'$ if $I \subseteq I'$ and $I \neq I'$. Within this work we also refer to $I.IN$ by the positive memory and with negative memory we refer to $I.OUT$. We say I is consistent iff $I.IN \cap I.OUT = \emptyset$. Note that the second component of I is only needed for the implementation part, for the remainder of this chapter we identify the partial interpretation I with $I.IN$ and assume $I.OUT = \emptyset$.

Definition 2.1.4. *An interpretation I models a rule $r \in P$, written $I \models r$, iff $head(r) \in I$ or $body^+(r) \nsubseteq I$ or $body^-(r) \cap I \neq \emptyset$. I models P, written $I \models P$, iff I models all rules of $gr(P)$.*

Now in order to determine the answer sets of P, we use the Gelfond-Lifschitz reduct:

Definition 2.1.5. *For any consistent interpretation M of P, let P^M be the program obtained from $gr(P)$ by deleting*

- *each rule that has a negative literal* $not\ B$ *in its body with* $B \in M$*, and*
- *all negative literals in bodies of the remaining rules.*

M is an answer set of P iff $M \models P^M$ and there exists no interpretation N of P such that $N \models P^M$ and $N \subset M$.

Example 2.1.3. *So for our example program* P *let I be the following interpretation:*

$$I = \{bird(tweety), penguin(nora), fly(tweety), bird(nora)\}$$

Then we get the the following reduction of P:

$$bird(tweety) : -penguin(tweety). \tag{2.13}$$
$$bird(nora) : -penguin(nora). \tag{2.14}$$
$$fly(tweety) : -bird(tweety). \tag{2.15}$$
$$bird(tweety). \quad penguin(nora). \tag{2.16}$$

We can see that the facts of P as well as the positive rules are not modified within the reduct. So for any positive program P, P^M *equals* $gr(P)$*. Only the non-monotonic rules change. In this case the non-monotonic rule* $fly(nora) : -\, bird(nora), not\, penguin(nora).$ *is deleted, because penguin(nora) is in I. The other non-monotonic rule* $fly(tweety) : -\, bird(tweety),$ $not\, penguin(\, tweety).$ *is transformed into a positive one, by deleting it's negative body. As one can see I is a minimal Herbrand model of P, and therefore an answer set to P.*

2.2 Grounding for Solvers

Conceptually, the approach for solving ASP programs is to check all interpretations that are possible answer sets. Common solvers like DLV [18] and clasp [8] follow a two step approach. First a component called the $grounder$ grounds the input program. After that, a second component called the $solver$, then is in charge of actually solving $gr(P)$. From this two step approach one problem arises: the potential exponential blowup of $gr(P)$. Because neither rules nor atoms are of limited size, there can be many variables within a rule. If we now ground a rule containing 10 variables, for a program containing 10 constants we obtain $10^{10} = 10.000.000.000$ ground rule instances. This can be avoided for many cases by intelligent grounding, which is used by most competitive ASP solvers nowadays.

Intelligent Grounding

Intelligent grounding builds on the idea of splitting the input program in several subprograms called modules, which are then grounded for a smaller calculated sub-domain. For this purpose a dependency graph and a component graph are built. From these graphs the strongly connected components (SCC) are extracted and ordered after their dependencies. The rules corresponding to each SCC are now called modules and can be evaluated separately in the obtained ordering. The grounding then starts at the lowest SCC and builds the set of significant atoms S. S contains all facts of P at start up. Rules are grounded in such a way that only those ground instances are created, where the grounded atoms of a rule are in S. Wherever a rule r is grounded its grounded head is added to S. To further optimize the grounding, grounded rules are simplified, by deleting atoms that are already true. For a more detailed description on intelligent grounding see [6] or [10], for a description on two state-of-the-art intelligent grounders used by DLV [18] and clasp [8].

While this optimization works quite well for most problem types, there are also problems where there are only a few huge SCCs or in the worst case only one SCC containing the whole program. In this case the restriction on the sub-domain is rather weak or not working at all.

Grounding on-the-fly

A rather new approach to solve ASP programs is called *grounding on-the-fly*. Here we only have a solver component that does the grounding itself, and only when it is needed. Since the grounding and solving step are intertwined, we know exactly what ground rules will be needed next, since we know the actual partial interpretation the solver is in, therefore only grounding those rules that can be grounded with that partial interpretation. Because of this, solvers using grounding on-the-fly are able to solve some ASP programs where other solvers fail, because the latter run out of space, due to the pregrounding.

Example 2.2.1. *The following example is a variation of the cutedge problem. A graph G is given. From G one edge is removed, then reachability is calculated on the remaining graph.*

$$delete(X,Y) :- edge(X,Y), not\, keep(X,Y). \tag{2.17}$$
$$keep(X,Y) :- edge(X,Y), delete(X1,Y1), X1! = X. \tag{2.18}$$
$$keep(X,Y) :- edge(X,Y), delete(X1,Y1), Y1! = Y. \tag{2.19}$$
$$reachable(X,Y) :- keep(X,Y). \tag{2.20}$$
$$reachable(X,304) :- reachable(X,Z), reachable(Z,304). \tag{2.21}$$

Reachability is easily solved by standard solvers (since reachability is a positive program, the intelligent grounder does not need to ground a single unnecessary rule). In contrast to that, they have huge troubles with this problem. The grounder does not know which edge is going to be deleted, therefore does not know which atoms are there, and therefore is not able to make any optimization on the grounding of the reachability rules. Therefore, a large number of ground rules is created that is not needed in the solving step. A grounding on-the-fly solver on the other side, has no problem with this program, since after one edge is deleted from the graph, the rest of the program is positive, and therefore easy to solve by simple propagation of rules.

We are only aware of two solvers that are based on grounding on-the-fly, namely ASPERIX [16] [17], and GASP [27].

2.3 Asperix

ASPERIX is a state-of-the-art solver that is based on grounding on-the-fly; it was first presented in [16] and [17]. The OMiGA solver presented in this work also uses grounding on-the-fly and is built on the basic concept used in ASPERIX. In this section we introduce some definitions used in ASPERIX, which we need for our algorithm.

Definition 2.3.1. *Let P be an ASP program and X be an interpretation, then the T_P operator is defined as follows:*

$$T_P(X) = \{a \mid \exists r \in gr(P), head(r) = a, body^+(r) \subseteq X, body^-(r) \cap X = \emptyset\}$$

Definition 2.3.2. *Let P be an ASP program. A computation for P is a sequence $\langle X_i \rangle_{i=0}^{\infty}$ of ground atom sets that satisfies the following conditions :*

- $X_0 = \emptyset$
- *(Revision)* $\forall i \geq 1, X_i \subseteq T_p(X_{i-1})$
- *(Persistence of beliefs)* $\forall i \geq 1, X_{i-1} \subseteq X_i$
- *(Convergence)* $C_\infty = \bigcup_{i=0}^{\infty} X_i = T_p(X_\infty)$
- *(Persistence of reasons)* $\forall i \geq 1, \forall a \in X_i \setminus X_{i-1}, \exists r_a \in gr(P)$ *s.t.* $head(r_a) = a$*, and* $\forall j \geq i-1, body^+(r_a) \subseteq X_j, body^-(r_a) \cap X_j = \emptyset$

The benefit of this approach is that the computation is based on forward chaining of rules in terms of the T_P operator. The answer sets are built incrementally and a complete grounding at start up is not needed. Therefore it is well suited to deal with first order rules while instantiating them during computation. Note that for this to work, all rules of P have to be safe. The following theorem shows that every existing sequence like this leads to an answer set.

Theorem 2.3.3. *Let P be a normal logic program and X be an ground atom set. Then, X is an answer set of P iff there is a computation $X_{i=0}^{\infty}$ for P such that $X_\infty = X$.*

For a proof of the above theorem please see [16]. Since the ASPERIX computation is based on the sequence given above and OMiGA uses a similar sequence, we use this theorem to prove correctness and completeness of our algorithm in Section 3.2. Furthermore we use the following notions from ASPERIX:

Definition 2.3.4. *Let r be a ground rule and $I = \langle IN, OUT \rangle$ a partial interpretation. We say that*

- *r is supported w.r.t. I when $body^+(r) \subseteq IN$,*
- *r is unsupported w.r.t. I when $body^+(r) \cap OUT \neq \emptyset$,*
- *r is blocked w.r.t. I when $body^-(r) \cap IN \neq \emptyset$,*

- *r is unblocked w.r.t. I when body*$^-(r) \subseteq OUT$*,*
- r *is applicable w.r.t. I when* r *is supported and not blocked i.e. body*$^+(r) \subseteq IN$ *and body*$^-(r) \cap IN = \emptyset$*.*

Not that not blocked is different from unblocked and that the set of not blocked rules is a super set of the set of unblocked rules of P.

Example 2.3.1. *Let* r *be the following rule:*

$$\mathit{fly}(tweety) :- \mathit{bird}(tweety), \mathit{not}\, \mathit{penguin}(tweety).$$

And let $I_1 = \langle \{bird(tweety)\}, \emptyset \rangle$*,* $I_2 = \langle \{bird(tweety)\}, \{penguin(tweety)\} \rangle$*,* $I_3 = \langle \emptyset, \{bird(tweety)\} \rangle$ *and* $I_4 = \langle \{bird(tweety), penguin(tweety)\}, \emptyset \rangle$*, then*

- *r is supported and not blocked, therefore applicable w.r.t.* I_1
- *r is supported and unblocked, therefore not blocked and applicable w.r.t.* I_2
- *r is unsupported w.r.t.* I_3
- *r is blocked w.r.t.* I_4

2.4 Rete network

The main problem of the ASPERIX approach is the evaluation of applicable nonground rule instances. In ASPERIX this evaluation happens at each calculation step, resulting in a recalculation of these rules all the time. In order to address this problem OMiGA uses a Rete network [7] to store the grounded atoms, as well as partial ground rule interpretations for later use. Rete networks are often used in large scale expert systems and rule based production systems, where they lead to an increase of speed of several magnitudes in comparison to a naive approach, because the Rete network performance is theoretically independent of the number of rules in the system.

A Rete network is a graph like structure consisting of intertwined nodes, where each node corresponds to a pattern occurring in the body of a rule. Each node has its own memory storing those ground atoms or partial rule interpretations corresponding to the pattern represented by that nodes.

Nodes that represent single atoms or partial rule interpretations are joined together and build new nodes until a leaf node is created which represents a complete rule interpretation. The path from a root node, which stores instances for single atoms, to a leaf node defines the join order of a complete rules body. If an instance is pushed into a leaf node, then this means a complete ground rule interpretation for the corresponding nonground rule has been found, and the grounded rule can be fired. For more details on how to represent an ASP program as a Rete network see Chapter 4.

The benefit of using a Rete network is to avoid the recalculation not only of complete ground rules but of partial rule interpretations as well, since all of those are stored within the joined nodes. Therefore a Rete network is a trade off between runtime and space usage since more memory is needed to store all the partial rule interpretations but almost no recalculation has to be done. Note that for OMiGA we decided to implement our own Rete network which is described in Chapter 5. The evaluation of OMiGA (Chapter 6) shows that Rete leads to a very good propagation speed and therefore indeed is a great tool for handling ASP rules.

CHAPTER 3

The OMiGA Algorithm

This chapter presents the algorithm underlying the OMiGA solver and is dedicated to prove that the algorithm is sound and complete. For this purpose we will first give the intuition of OMiGA's base functions and then present the algorithm behind OMiGA. Then within Section 3.4 we present the formal definition of these functions and the proof that the presented algorithm is indeed correct. Finally we give a complexity analysis determining an upper bound for the runtime.

3.1 Intuition

The OMiGA algorithm uses grounding on the fly to generate ground rule instances during solving. For this purpose a Rete network is used to store all grounded atoms, to find all applicable nonground rule instances and to find all those ground rule instances on which OMiGA can still guess. The Rete network therefore represents our knowledge base and all functions that our algorithm uses are somehow based on the Rete network.

The **prop** function is in charge of propagating knowledge through the Rete network and is the only function that is capable of generating new positive knowledge by applying supported unblocked rules. One call to prop triggers a propagation within our Rete network such that all supported unblocked rules are fired until a fixpoint is reached. (So prop does not only apply one such rule but rather after a call to prop there is no supported unblocked rule left that has not yet been applied.)

In order to deal with non-monotonic rules, OMiGA guesses on supported not blocked rules. For this purpose the **choice** function is implemented. A call to choice triggers a guess on one supported not blocked ground rule instance. Note that those ground rule instances are not evaluated by the choice function itself but are gathered during the propagation phase and stored in so called choice nodes within the Rete network. The choice function simply takes one such ground rule instance from a choice node and either performs a positive or a negative guess for that rule. Which kind of guess is made is determined by a boolean parameter of this function. If a positive guess is made the choice function puts the grounded atoms of the negative body of the chosen rule into the Rete networks negative memory. If a negative guess is made the choice function simply puts the grounded chosen rules head into the Rete networks negative memory. (Note that since we only make choices on supported rules, and since all rules must be safe, we can indeed ground the negative body of such a rule, since all variables are already bound by the positive body). Note that the choice function only adds things into the negative memory and the positive memory never changes. Observe that in case of a positive guess the head of the chosen rule will be added into the positive memory by the next call to the prop function, since the formerly supported, not blocked rule became supported and unblocked because of the positive guess.

Furthermore note that the choice function will always return true when doing a positive or negative guess. The only case in which choice returns false is when there are no more supported, not blocked rules left.

When guessing on rules or when applying rules it might happen that an inconsistency arises, meaning that either a constraint is violated or the same grounded atom is added to the positive as well as to the negative memory. In both cases further calculation becomes unnecessary, since inconsistency cannot be revoked by further propagation or choice operations. For this purpose we have the inconsistent function which checks whether the current interpretation stored in the Rete network is consistent. The inconsistent function will return true if an inconsistency was detected and false otherwise.

In case of contradiction or in case of no more possible guesses, we have reached a point where no further call to *choice* or *prop* will change the current interpretation. Therefore we have to backtrack to a previous choice and make a different guess on a previous chosen rule. For this purpose a call to the *backtrack* function returns the Rete network to the state it was in before the last positive guess (i.e. returns the partial interpretation before that choice). For this purpose OMiGA utilizes two global stacks. One for storing the interpretations before doing a guess, and one for storing the ground rule instances OMiGA guessed on. A call to backtrack pops elements from both stacks until the desired interpretation is reached. The last rule of the rule-stack will be used in the next choice by doing a negative guess. In case of no positive guess being on the stack backtrack returns $\emptyset$. Note that within the OMiGA algorithm there is a flag variable called *nextAlternative*. This variable is always false except after a call to backtrack which will set this variable to true. This variable is used as a parameter for the choice function in order to indicate that a positive or negative guess should be done. Observe that OMiGA therefore first checks for positive guesses. So the first answer set looked at is the one with all possible rules guessed positively while the last answer set being checked is the one with all rules being guessed negatively.

Finally a run of the whole OMiGA algorithm looks like this: First the input program is read and rewritten into a program where all non-monotonic rules have unique heads. From this rewritten program a Rete network is built and all facts of the input program are pushed into the Rete network. Then $Solve(P)$ 3.1 is called, which does all the work for calculating the answer sets of the input program, by alternating between a call to choice and prop. In case of contradiction or a found answer set backtrack is called and further choices and propagation is done until all possible answer sets have been checked.

3.2 The algorithm behind OMiGA

```
Require: All rules of $P$, with a non empty negative body have unique heads.
Ensure: $AS$ contains all answer sets of $P$.
 1: $finished$ = false
 2: $AS = \emptyset$
 3: $nextAlternative$ = false
 4: $I = \langle\emptyset,\emptyset\rangle$
 5: $I$ = prop($I, P$);
 6: while !$finished$ do
 7:   $I_{guess}$ := choice($I$,$nextAlternative$,$P$) // try to guess
 8:   if $I_{guess}! = \emptyset$ then
 9:     // check if guess was possible
10:     $I = I_{guess}$
11:     $nextAlternative$ = false // next guess is positive
12:     $I$ = prop($I$,$P$)
13:     if inconsistent($I$,$P$) then
14:       $I$ = backtrack() // contradiction: stop calculation of current branch
15:       if $I \neq \emptyset$ then
16:         $nextAlternative$ = true // not finished yet, next guess is negative
17:       else
18:         $finished$ = true // no more alternatives
19:       end if
20:     end if
21:   else
22:     // No more guess in current branch
23:     $AS = AS \cup \{I.IN\}$ // An answer set was found
24:     $I$ = backtrack()
25:     if $I \neq \emptyset$ then
26:       $nextAlternative$ = true // not finished yet, next guess is negative
27:     else
28:       $finished$ = true // no more alternatives
29:     end if
30:   end if
31: end while
32: return $AS$
```

Algorithm 3.1: Solve(P)

The solve procedure first initializes its variables (line 1 to 4). The boolean flags *finished* and *nextAlternative* are set to false. While *finished* is used to signal that the algorithm is *finished*, *nextAlternative* is used to call choice (line 7), and indicates if a positive or negative guess is made next. Initially the interpretation is set to $\langle \emptyset, \emptyset \rangle$ (line 4), meaning that OMiGAs positive and negative memory do not contain any facts. In line 5 the call to prop() ensures that everything that can be derived for sure without any guessing is now within the positive memory. Note that if the input program was a positive program the only answer set is found at the end of this prop call.

The main part of this algorithm is the wile loop from line 6 to 31, which runs for as long as our algorithm is not finished. Within this while-loop we make a choice (line 7). Note that every successful choice is followed by a call to prop in line 12. Whenever choice returns false (i.e. there are no more rules to guess on left), the else branch (line 21 to 30) is entered. Here we add the current interpretation into our return set (line 23). Note that the current interpretation at this point must indeed be an answer set since there are no more rules left to guess on and no contradiction was found. Observe that the inconsistency check is done in the if branch in line 13 to 20. So if the else branch is entered the current interpretation has already been checked for inconsistency in the last iteration of the while loop. In line 24 backtrack is called which will return the current interpretation to its state before the last positive guess. In case backtrack returns $\emptyset$ (i.e. all possible answer sets have been checked) finished is set to true and the while loop terminates. Finally in line 32 all answer sets are returned and the calculation is finished.

Note that *nextAlternative* is always set to false after choice was called successfully (line 11), and always set to true after a call to backtrack() (line 16 and 26). This means that unless backtrack was called after the last guess, the next guess will be a positive one, meaning that positive guesses are done before negative ones.

Example 3.2.1. *Let P be the following ASP input program for the $Solve$ procedure.*

$$animal(sara). \quad bird(tweety). \quad penguin(nora).$$

$$fly(X) :- bird(X), not\, penguin(X). \tag{3.1}$$

$$bird(X) :- penguin(X). \tag{3.2}$$

$$bird(X) :- animal(X), not\, noBird(X). \tag{3.3}$$

$$noBird(X) :- animal(X), not\, bird(X). \tag{3.4}$$

Intuitively here we got three animals called tweety, nora and sara. We know that tweety is a bird and nora is a penguin. We furthermore have knowledge that birds usually fly except for penguins which do not fly. Finally we know that each animal is either a bird or not. The aim of P is to calculate which of our animals can fly, and which of our animals are birds.

Solve initializes its data structures and then calls prop in line 5. Note that before solve was called the Rete network has already been initialized. Therefore the facts are already in the Rete queue waiting to be propagated. So when prop is called all those facts are saved within the Rete *memory, one at a time. (Note that OMiGA follows some order based on hash values. For better readability we will consider them in order of the program (from left to right and top to bottom)).*

First animal(sara) is saved into the Rete network. Since animal is only involved in rule 3.3 and 3.4, and since there is nothing in our negative memory at the moment nothing is derived from this. Next bird(tweety) is added. This does not derive anything either since bird is only involved positively in rule 3.1. Next penguin(nora) is added into the Rete network which triggers rule 3.2 to fire and derives bird(nora). Finally bird(nora) is added into the Rete network, which does not trigger further derivations and therefore the call to prop terminates. While our negative memory is still empty the positive memory now contains:

$$\begin{aligned} animal(sara).& \quad bird(sara). \\ bird(tweety).& \quad penguin(nora). \quad bird(nora). \end{aligned}$$

We enter the while loop and do the first guess (line 7). Note that penguin does not occur in any head while bird and noBird do. Therefore OMiGA will consider rule 3.1 first when guessing. Furthermore rather than guessing on rule 3.1, OMiGA will do an optimization at this point and close the SCC containing penguin, meaning that from this point on OMiGA knows that no other penguin exists except those that are already in the positive memory. Therefore penguin(tweety) and penguin(sara) can now be considered to be in the negative memory (although this does not need additional space).

Since the SCC is now closed the call to prop in line line 12 has to work on bird(tweety) since we now know that tweety is no penguin. Therefore fly(tweety) is derived. No inconsistency is detected (line 13) and we re-enter the while-loop, again doing a guess in line 7.

A guess is made on the ground rule bird(sara):-animal(sara), not noBird(sara). Since no back-track has been issued so far a positive guess is made and noBird(sara) is added into the Rete network queue for the negative memory. The call to prop in line 12 now saves noBird(sara) into the negative memory which leads to the fulfillment of rule 3.3 with X = sara. Therefore bird(sara) is derived and added to the Rete network. This again triggers the derivation of fly(sara) because of rule 3.1 (since the penguin SCC is already closed, and no penguin named sara is known). Finally fly(sara) is added to the Rete network and propagation stops. No contradiction is detected in line 13. Our knowledge base now looks like this:

$$\begin{aligned} animal(sara).& \quad bird(sara). \\ bird(tweety).& \quad penguin(nora). \quad bird(nora). \\ fly(tweety).& \quad fly(sara). \\ -penguin(tweety).& \quad -penguin(sara). \\ -noBird(sara).& \end{aligned}$$

Within the next iteration, choice returns false, since there are no further supported not blocked rules left. (Note that rule 3.4 is only supported for X = sara, because of animal(sara), but is also blocked because of bird(sara)). The current interpretation is added into our return set AS, and backtrack is called returning to a state before the guess of rule 3.3 with X = sara. NextAlternative is set to true and our knowledge base looks like this:

$$\begin{aligned} animal(sara). \quad & penguin(nora) \quad bird(nora).\\ bird(tweety). \quad & fly(tweety).\\ -penguin(tweety). \quad & -penguin(sara). \end{aligned} \tag{3.5}$$

In the next iteration, choice (line 7) will now do a negative guess on bird(sara):-animal(sara), not noBird(sara)., therefore pushing bird(sara) into the negative memory. The call to prop (line 12) adds bird(sara) into the negative memory which triggers rule 3.4 with X = sara, leading to the derivation of noBird(sara), which again is added into the Rete network, but does not trigger further rules. Our knowledge base looks like this:

$$\begin{aligned} animal(sara). \quad & penguin(nora) \quad bird(nora).\\ bird(tweety). \quad & fly(tweety).\\ noBird(sara). \quad & \\ -penguin(tweety). \quad & -penguin(sara).\\ -bird(sara). \quad & \end{aligned} \tag{3.6}$$

Since no more supported not blocked rules remain, choice (line 7) will again return false and the else branch is entered. We add the current interpretation to AS (line 23) and trigger backtrack, which returns $\emptyset$ since no more positive guess is on the stack (rule 3.3 with X=sara was guessed negatively. Rule 3.4 has been triggered by propagation and rule 3.1 was solved by closing an SCC which does not count as a positive guess as well). Therefore finished is set to true in line 28 and the while loop ends. Finally AS is returned and the calculation terminates. Note that AS now contains two interpretations (A_1 and A_2) which both are correct answer sets of the input program:

$$\begin{aligned} A_1 = \{animal(sara). \quad & bird(sara).\\ bird(tweety). \quad & penguin(nora). \quad bird(nora).\\ fly(tweety). \quad & fly(sara).\\ -penguin(tweety). \quad & -penguin(sara).\\ -noBird(sara).\} \quad & \end{aligned}$$

$$\begin{aligned}A_2 = \{&animal(sara). \quad penguin(nora). \quad bird(nora).\\ &bird(tweety). \quad fly(tweety).\\ &noBird(sara).\\ &-penguin(tweety). \quad -penguin(sara).\\ &-bird(sara).\}\end{aligned}$$

3.3 Formal Definitions

Note that for OMiGA the input program P must be safe and that OMiGA only deals with programs where all contained non-monotonic rules have unique heads. This is achieved by rewriting P before solving it. For a detailed description of our rewriting see Section 5.2. In the following we define OMiGA's base functions and structures which are later used to prove correctness. The base functions of OMiGA are:

- $prop$, for propagation,
- $choice$, for guessing on one rule,
- $backtrack$, for restoring a previous interpretation, and
- $inconsistent$, for determining if the current interpretation is inconsistent.

The prop function relies on a least-fixpoint calculation which uses an immediate-consequence operator as follows:

Definition 3.3.1. *Given a program P, the* immediate-consequence operator $\Gamma_P : \mathcal{I} \to \mathcal{I}$ *is defined as:*

$$\Gamma_P(\langle In, Out\rangle) = \langle In \cup \{head(r) \mid r \in gr(P) \text{ and } \langle In, Out\rangle \models r\}, Out\rangle$$

Observe that Γ_P is monotonic and $\mathcal{I}$ is finite. Given an interpretation $I \in \mathcal{I}$, we define the iterated application of Γ_P starting at I as follows:

$$\begin{aligned}\Gamma_P^0(I) =& I\\ \Gamma_P^{n+1}(I) =& \Gamma_P(\Gamma_P^n(I))\end{aligned}$$

Since $\mathcal{I}$ is finite, we do not need consider transfinite iteration here.

Definition 3.3.2. *For an interpretation I, the least-fixpoint of Γ_P starting at I is $lfp_P^I = \Gamma_P^n(I)$ such that $n \in \mathbb{N}$ is the smallest number with $\Gamma_P^{n+1}(I) = \Gamma_P^n(I)$.*

Note that by monotonicity of Γ_P and finiteness of $\mathcal{I}$, the least-fixpoint lfp_P^I exists and is reachable in finitely many steps. We can now define the propagation function as follows:

Definition 3.3.3. *$prop : \mathcal{I} \times \mathcal{P} \to \mathcal{I}$ is a function leading to a new interpretation $prop(I, P) = \langle lfp_P^I.IN, I.OUT\rangle$.*

Intuitively, the propagation function applies supported unblocked rules, by adding their head into the positive memory, until no more such rule exists. Observe that after a propagation, there is no supported unblocked rule $r \in P$ left, that is applicable with respect to I_{i+1}, since the head of all such rules has been added to $I_{i+1}.IN$. Furthermore, observe that only the positive memory may increase i.e. $IN_i \subseteq IN_{i+1}$, while the negative memory always stays the same i.e. $OUT_i = OUT_{i+1}$.

Lemma 3.3.4. *Let A be an answer set of P and $I.IN \subseteq A$. Then $prop(I,P) \subseteq A$.*

Proof. $prop(I,P)$ only applies supported unblocked rules of P with respect to I. Those rules must also be supported and unblocked with respect to. A. Any answer set must contain head(r) for each supported unblocked rule $r \in P$ with respect to A. Therefore $prop(I.IN, P) \subseteq A$. $\square$

Definition 3.3.5. *A stack over a set T is a sequence $S = \langle t_1, t_2, ..., t_n \rangle$ where $t_i \in T$ for $1 \leq i \leq n, n \geq 0$. We use push, peek and pop in the usual way, i.e. S.pop$() = \langle t_1, t_2, ..., t_{n-1} \rangle$, S.peek$() = t_n$, and S.push$(t) = \langle t_1, t_2, ..., t_n, t \rangle$ for $t \in T$. In case that there are no elements on the stack S.peek() returns $\emptyset$ and s.pop() returns an empty stack $\langle\rangle = nil$. Within the OMiGA algorithm the two following global stacks are used:*

- *SR is a stack of rules*
- *SI is a stack of interpretations*

Intuitively SR contains all the rules OMiGA has guessed on in order to reach the current interpretation, in the order they were guessed on. SI on the other side contains the partial interpretations that were reached with the corresponding guess on the rule of SR. Both stacks can only change through push operations from the choice method or pop operations from the backtrack method. Both stacks are always changed at the same time to guarantee that SI_i corresponds to SR_i. By $\mathcal{SI}$ (resp. $\mathcal{SR}$) we refer to the set of all stacks of rules (resp. interpretations).

Definition 3.3.6. *$choice : \mathcal{I} \times \mathcal{B} \times \mathcal{P} \times \mathcal{SR} \times \mathcal{SI} \rightarrow \mathcal{I} \times \mathcal{SR} \times \mathcal{SI}$ is a function leading to a new interpretation $choice(I, B, P, SR, SI) = \langle I', SR', SI' \rangle$, where I is the actual interpretation, B is a boolean flag, P is the input program and SR, SI are our two global stacks. Furthermore,*

- *for B = true, $I' = \langle I.IN, I.OUT \cup head(r) \rangle$, where $r = SR.peek()$. $SR' = SR$ and $SI' = SI.push(I')$. If no such rule exists $I' = \emptyset$, $SR' = SR$ and $SI' = SI.push(I')$.*
- *for B = false, $I' = \langle I.IN, I.OUT \cup body^{-}(r) \rangle$, where r is the lexicographic next supported not blocked rule and $r \notin SR$. $SR' = SR.push(r)$ and $SI' = SI.push(I')$. If no such rule exists, then $I' = \emptyset$, $SR' = SR$ and $SI' = SI.push(I')$.*

Intuitively, the choice function chooses one supported not blocked rule, and either puts its head or its negative body into the negative memory. Which of these options is chosen depends on the boolean flag parameter. Observe that only the negative memory is changed by a call to choice, while the positive memory stays unchanged. Furthermore, if the interpretation returned by $choice(I, true, P, SR, SI)$ equals $\emptyset$ this means that there are no more supported not blocked rules left to guess on. If the interpretation returned by $choice(I, false, P, SR, SI)$ equals $\emptyset$ this means that there is no rule we have guessed on, for which we have not guessed positively and negatively. So if $choice(I, true, P, SR, SI)$ and choice$(I, false, P, SR, SI)$ return $\emptyset$, this indicates that the calculation has checked all possible choices. Note that when we call $choice$ within the algorithm we omit the SR and SI parameters, since in the implementation this is a global stack.

Definition 3.3.7. *$backtrack : \mathcal{SI} \times \mathcal{SR} \rightarrow \mathcal{I} \times \mathcal{SR} \times \mathcal{SI}$ is a function restoring a previous interpretation $backtrack(SI, SR) = \langle I', SR', SI' \rangle$, such that $I' = SI_i$ for the highest integer i, for which SR_i is a rule that was guessed positively. We pop every element with a higher index than i from the stacks i.e. $SR' = SR.pop()^{j-i}$ and $SI' = SI.pop()^{j-i}$, where j is to the size of the stacks.*

Intuitively backtrack resets OMiGA to the interpretation before the last positive guess. Note that for the algorithm we omit the stack parameters since those are global stacks within the actual implementation, that keep track of the guessing and backtracking within OMiGA.

Definition 3.3.8. *$inconsistent : \mathcal{I} \times \mathcal{P} \rightarrow \mathcal{B}$ is a function such that $inconsistent(I, P) = 1$ if $I.IN \cap I.OUT \neq \emptyset$ or some $r \in gr(P)$ exists such that r is a constraint rule and r is supported and unblocked with respect to. I. Otherwise inconsistent(I,P) = 0.*

Definition 3.3.9. *A computation tree T for a program P is a tree where each node $n \in T$ is a triple $\langle I, SR, SI \rangle$ and the following holds:*

- *root of $T = \langle \langle \emptyset, \emptyset \rangle, nil, nil \rangle$, having exactly one child : $\langle prop(prop(\langle \emptyset, \emptyset \rangle, P), P), nil, nil \rangle$*
- *if $\langle I, SR, SI \rangle$ is a non-root node of T and if $choice(I, false, P, SR, SI) \neq \emptyset \neq choice(I, true, P, SR, SI)$, then $\langle I_l, SR_l, SI_l \rangle = choice(I, false, P, SR, SI)$, $\langle I_r, SR_r, SI_r \rangle = choice(I, false, P, SR, SI)$ and this node has two children: $\langle prop(\ choice(I, true, P, SR, SI), P), SR_l, SI_l \rangle$ and $\langle prop(choice(I, false, P, SR, SI), P), SR_r, SI_r \rangle$.*
- *nothing else is in T*

Definition 3.3.10. *A* computation sequence *$S = \langle S_0, S_1, ..., S_{max} \rangle$ of OMiGA is defined as the path from the root node of a computation tree to one of its leaf nodes.*

By $S_i.I$ we refer to the first part, the interpretation, of the i-th tuple within the Sequence S while by $S_i.SR$ (resp. $S_i.SI$) we refer to the corresponding stack SR (resp. SI) of that node. By S_{max} we denote the last element of S.

Note that whenever $backtrack(SR, SI)$ is called, the algorithm jumps (with the next choice) to another branch in the computation tree. So when speaking of one computation sequence one is only interested in the lines 1-12 and line 23. Everything else is code for backtracking due to either an answer set or a contradiction being reached.

3.4 Correctness

This section is dedicated to prove correctness for the algorithm presented in Figure 3.1. For this purpose we will first proof soundness and then completeness. Note that for this prove we will only consider a bounded input domain, although OMiGA can deal with unbounded input domains (generated by operators or function terms), but then termination is not guaranteed.

Soundness

Theorem 3.4.1. *All interpretations returned by $Solve(P)$ are answer sets.*

Proof. Let $A \in AS$, where $AS = Solve(P)$, and let S be the computation sequence according to A. Then to show soundness we show that A is an answer set by showing that S satisfies the properties of definition 2.3.2. Note that for readability we write X_i to denote $S_i.I.IN$.

- $X_0 = \emptyset$ holds since OMiGA starts with $I.IN = \langle \emptyset, \emptyset \rangle$.
- To show that for all $i \geq 1, X_i \subseteq T_P(X_{i-1})$, observe that the only method changing $I.IN$ within a sequence of OMiGA is the $prop(I, P)$ function which only applies supported unblocked rules. The TP operator returns the set of all heads of all supported not blocked rules. The set of supported unblocked rules is a subset of supported not blocked rules, since every unblocked rule is also not blocked. Therefore this property holds.
- For all $i \geq 1, X_{i-1} \subseteq X_i$ holds, since the only function changing $I.IN$ is the prop(I, P) function, which only adds atoms but never removes them.
- $\bigcup_{i=0}^{\infty} X_i = T_P(X_\infty)$: S is monotone since atoms are only added but never removed from $I.IN$. Therefore it suffices to show that $S_{max}.I.IN = T_P(\ S_{max}.I.IN)$. This holds, since in order to reach line 23, choice returned $\emptyset$, indicating that there are no more applicable rules, that have not yet been applied. This means that $T_P(S_{max}.I) \subseteq S_{max}.I.IN$ since the heads of all applicable rules are already in $S_{max}.I.IN$ and $S_{max}.I.IN \subseteq T_P(S_{max}.I.IN)$ holds since only propagate adds into OMiGA's $I.IN$. Therefore any fact in $S_{max}.I.IN$, was derived by an applicable rule, and therefore is also part of $T_P(S_{max}.I.IN)$.
- For all $i \geq 1, \forall a \in X_i \setminus X_{i-1}, \exists r_a \in P s.t. head(r_a) = a$, and $\forall j \geq i-1, body^+(r_a) \subseteq X_j, body^-(r_a) \cap X_j = \emptyset$ holds since the only function changing $S_i.I.IN$ is the $prop(I, P)$ function, by adding heads of supported unblocked rules. Since S is monotone nothing is removed from it, resulting in any supported unblocked rule r that holds with respect to $S_i.I$ also holds within $S_{i+1}.I$, since $S_i.I \subseteq S_{i+1}.I$. Note: should some $a \in body^-(r)$ of a supported unblocked rule appear in $S.I.IN$, a contradiction is detected (line 12).

Let P be an ASP program of bounded input domain, then $gr(P)$ is bounded as well and S can only be of size at most $|gr(P)| + 2$ and OMiGA terminates. Note that OMiGA might not terminate for programs of unbounded domain. (This can happen due to operators, or function terms). Since for ASP programs of bounded domain OMiGA terminates, S fulfills the above properties and because of Theorem 2.3.3 all interpretations AS returned by OMiGA are answer sets. □

Completeness

To show completeness we first show that any answer set of a Program P has a corresponding computation sequence, by constructing such a sequence for an arbitrary answer set of P. Then we show that OMiGA checks at least one such corresponding computation sequences for every possible answer set.

Lemma 3.4.2. *For every answer set A of the input program P, there exists some computation sequence S^A.*

Proof. Let A be an arbitrary answer set of an ASP program P. (Recall that for this proof we assume that all rules with negative body have unique heads). Then we construct a computation sequence $S^A = \langle S_0, S_1, ..., S_{max-1}, S_{max}\rangle$ as follows: $S_0 = \langle\langle\emptyset,\emptyset\rangle, nil, nil\rangle$, $S_1 = \langle prop(\langle\emptyset,\emptyset\rangle, nil, nil\rangle$. Then for each S_i with i > 1, we choose the lexicographic next applicable rule $r \in gr(P)$ with respect to $S_{i-1}.I$, s.t. $r \notin S_{i-1}.SR$. If $head(r) \in A$ then we consider a positive guess i.e. $\langle I', SR', SI'\rangle = choice(S_{i-1}.I, true, P, S_{i-1}.SI, S_{i-1}.SR)$. Then $S_i = \langle prop(I', P), SR', SI'\rangle$. Otherwise we consider a negative guess i.e. $\langle I', SR', SI'\rangle$ $= choice(S_{i-1}.I, false, P, S_{i-1}.SI, S_{i-1}.SR)$. Then $S_i = \langle prop(I', P), SR', SI'\rangle$. Observe that S is indeed a computation sequence of T, since $|gr(P)|$ is finite and therefore, after finitely many steps, there is no more applicable rule $r \notin SR$ and the sequence ends with S_{max}. We show that $S_{max}.I.IN = A$:

- By induction on $i \geq 0$ we show that ($S_i.I.IN \subseteq A$), thus $S_{max} \subseteq A$:

  1. Induction base: $S_0.I.IN = \emptyset \subseteq A$.
  2. Induction step: For the induction step we now show: if $S_i.I.IN \subseteq A$ then $S_{i+1}.I.IN$ $\subseteq A$. If $i = 0$, then $S_1.I.IN = prop(\emptyset, P).I.IN \subseteq A$ (Lemma (3.3.4)). Otherwise ($i \geq 1$) we distinguish whether the head of the lexicographic next rule r is in A:
     - $head(r) \in A$, then $S_{i+1}.IN = prop(choice(S_i.I, true, P, S_i.SR, S_i.SI), P).IN$. For the Interpretation I' returned by $choice(S_i.I,\ true,\ P, S_i.SR, S_i.SI)$ we get $I'.IN \subseteq A$ since $head(r) \in A$ and $I' = \langle S_i.I.IN \cup head(r), S_i.I.OUT\rangle$. Note that by the induction hypothesis $S_i.I.IN \subseteq A$. From $I'.IN \subseteq A$ and Lemma (3.3.4) we get $prop(I', P) \subseteq A$, therefore $S_{i+1}.I.IN \subseteq A$.
     - $head(r) \notin A$, then $S_{i+1}.IN = prop(choice(S_i.I, false, P, S_i.SR\ , S_i.SI), P).IN$. For the Interpretation I' returned by $choice(S_i.I,\ false,\ P, S_i.SR, S_i.SI)$ we get $I'.IN \subseteq A$ since $I'.IN = S_{i-1}.I.IN$. Note that by the induction hypothesis $S_i.I.IN \subseteq A$. From $I'.IN \subseteq A$ and Lemma (3.3.4) we get $prop(I', P) \subseteq A$, therefore $S_{i+1}.I.IN \subseteq A$.

  Therefore it holds that $S_{max}.I.IN \subseteq A$.

- ($S_{max}.I.IN = A$): Since $S_{max} \subseteq A$ it follows that $gr(P)^A \subseteq gr(P)^{S_{max}}$. Since a reduct is a positive program it follows that if $S_{max} \models gr(P)^{S_{max}}$ then $S_{max} \models gr(P)^A$. Since A is a minimal model of $gr(P)^A$ and $S_{max} \subseteq A$ it then follows that $S_{max} = A$. Furthermore it holds that if $S_{max} \models gr(P)$ then $S_{max} \models gr(P)^{S_{max}}$. Therefore it only remains to show that $S_{max} \models gr(P)$.

  Towards contradiction assume there exists a rule r $\in gr(P)$ s.t. $body^+(r) \subseteq S_{max}.I.IN$ and $head(r) \notin S_{max}.I.IN$. If $body^-(r) \subseteq S_{max}.I.OUT$ then $head(r) \in prop(S_{max}.I, P).IN$, which contradicts that $head(r) \notin S_{max}$, because the prop function computes a fix point and $S_{max} = prop(choice(\ S_{max-1}.I, B, S_{max-1}.SR, S_{max-1}.SI), P)$, where B is the corresponding boolean value used to construct S_{max}. On the other hand if $body^{-}(r) \not\subseteq S_{max}.I.OUT$ then r is a supported not blocked rule which contradicts that S_{max} is the last element of the computation sequence S.

No such r exists and it follows that $S_{max} \models gr(P)$, and consequently $S_{max} \models gr(P)^{S_{max}}$. Thus $S_{max} \models gr(P)^{A}$ and therefore $S_{max}.I.IN = A$.

□

Lemma 3.4.3. *OMiGA traverses a computation tree T of P, reaching all leaf nodes of T that might correspond to an answer set of P.*

Proof. OMiGA starts with $\langle \emptyset, \emptyset \rangle$ (line 4), which corresponds to the root of T (S_0). Then propagate is called (line 5) corresponding to S_1. Then within the while loop (line 6 to 31) OMiGA alternates between choice and propagate calls until a call to choice returns $\emptyset$. Since choice only returns $\emptyset$ when there is no further applicable rule, this means for every applicable rule $r \in gr(P)$ a prop(choice()) has been made, meaning we finished a computation sequence and are at a leaf node within the computation tree. When choice returns $\emptyset$ backtrack() is called, reverting all previous guesses until the last positive guess. Since every rule is guessed positively first by choice() (this is guaranteed since $nextAlternative$ is always false except for an iteration directly after a call to backtrack() (line 16 & 25)), we start with the outermost left calculation sequence of the tree, and through backtrack check all the other sequences until we reach the right outermost one. The only reason why a leaf node might not have been reached by OMiGA is the premature call to backtrack (line 14) because an inconsistency was detected (line 13). But those sequences are not corresponding to any answer set, since answer sets need to be consistent. □

Theorem 3.4.4. *OMiGA returns all answer sets.*

Proof. By Lemma 3.4.2 there is a computation sequence for every answer set of P. From Lemma 3.4.3 we get that OMiGA checks any sequence S that can be built from T that might be an answer set. Since T contains all computation sequences, OMiGA therefore returns all answer sets of P. □

3.5 Runtime analysis

In this section we analyze the algorithm behind OMiGA in terms of runtime. For this purpose we will have a look at the Solve algorithm (3.1) and compute the runtime with respect to OMiGAs base functions. Then the runtime of all used functions is analyzed and finally a complete analysis is given. The aim of this analysis is to find an upper bound. Note that reading the input program and building the Rete network can be done efficiently in linear time and does not add to the worst case complexity of the problem.

In order to analyze the OMiGA algorithm in detail we will split it into three parts.

1. everything before the while loop

2. the if branch within the while loop
3. the else branch of the while loop

Line 1 to 5: Note that line 1 to 4 contain only initialization, which can be done in constant time. Therefore the first part is dominated by the runtime of the propagation function ($prop$), which is called in line 5.

While loop: In order to split the while loop into two parts observe that the else branch is only reached, if the choice function returns false. Furthermore recall that choice only returns false if there are no more non-monotonic ground rules left, that have not yet been considered. Therefore the if branch can be executed at most n times, where n is the number of non-monotonic ground rules of P, before the else branch is considered.

Furthermore observe that for the purpose of finding an upper bound we will consider a run where the code in line 14 - 18 is not reached, since this is an optimization, which detects inconsistency before reaching a complete interpretation - therefore reducing the number of guesses needed. Note that the code that is triggered by this optimization is actually the same as the code within the else branch. Therefore no benefits arise from considering a run where this code is not reached.

Note that within the else branch backtrack is called in line 24. The else branch will set $finished$ to true only if backtrack returns $\emptyset$, which by definition of backtrack will be only after all possible choice combinations have been considered. For this purpose backtrack has to be called at most 2^n times, since by definition backtrack does not consider any combination of choices twice. As an upper bound we therefore consider the else branch to be executed 2^n times. Furthermore observe that backtrack returns the interpretation to a previous state, therefore resetting 1 to n non-monotonic rules that need to be guessed on again.

Therefore we can consider the while loop being executed like this: 1 to n times a choice is made and the if branch is entered. Then an additional choice is made and the else branch is entered. This is done up to 2^n times since only after all combination of choices have been considered $finished$ is set to true and the while loop terminates. Therefore we get the following upper bound:

$$propagate + 2^n * ((n+1) * choice + n * \textit{if-branch} + \textit{else-branch})$$

Where

- $propagate$ refers to the runtime of the call of the propagate function in line 5
- $choice$ refers to the runtime of the call of the choice function called n line 7
- *if-branch* refers to the runtime of the the complete if branch from line 8 to 20

- *else-branch* refers to the runtime of the complete else branch from line 21 to 30

Note that the else branch is dominated by the backtrack function since the remaining parts of the else branch can be executed in linear time. Furthermore note that the *if-branch* contains two function calls that need further evaluation, namely the propagation function and the inconsistency function. Everything else within the if branch can be done in linear time. Now let $prop, choice, backtrack, inconsistent$ be the runtime of the functions propagate, choice, backtrack and inconsistent need to be executed respectively. Then we get the following upper bound for running OMiGA, in respect to the used functions:

$$prop + 2^n * ((n+1) * choice + n * (prop + inconsistent) + backtrack)$$

Note that in terms of O-notation we can drop the first $prop$ since $2^n * n * prop$ will definitely dominate it. Furthermore we can appraise n+1 with n, reducing our bound to:

$$2^n * (n * (choice + prop + inconsistent) + backtrack)$$

We now analyze the runtime of these functions in detail. For a deeper understanding on how they are actually implemented we refer to Chapter 5 (the System Architecture chapter).

choice: The choice function picks one non-monotonic ground rule and does either a positive or a negative guess for this rule. In order to pick such a rule, OMiGA has to consider, in the worst case, all memories storing such rules. The number of such memories coincides with the numbers of nonground non-monotonic rules of P.

In case of a negative guess, the head of the picked rule is added to the negative memory. In case of a positive guess, all atoms of the negative body are put into the negative memory. Storing an atom needs p time, where p is the number of variables of the nonground atom to be stored. We therefore get an upper bound for the choice function of $O(r * a * p)$, where r is the number of nonground rules of P, a is the number of atoms within the rule with the most atoms of P, and p is the arity of the predicate with highest arity of P.

inconsistent: The actual runtime of the inconsistent function is $O(1)$, since we only have to check a Boolean flag. This flag is actually set during the propagation process (so by the propagation function). The time needed for this is $O(a)$, where a is the number of atoms being added, since for any atom added we have to check if the counterpart is already in the negative memory, or vice versa. Since the propagation function has to consider every atom anyway and needs even much more time propagating them through the Rete network, we will not mention this, when analyzing the propagation function anymore.

prop: The propagation function adds atoms into the Rete network. For each atom joins are triggered in order to build partial and finally complete ground rule instances. For this purpose each atom can trigger up to $a - 1$ joins per rule it is involved in. (Of course in the worst case an atom can be involved in all nonground rules of P. The size of a joins result equals the number of ground instances that are stored in the memory with which the new atom is joined. In the worst case this is c^v, where c are all the constants of P (the active domain), and v is the number of variables occurring in P; resulting joined instances can be of size v and therefore need up to v time in order to be stored, which leads to an upper bound of $O((c^v)^{a-1} * v * r)$ for one atom being propagated, where r is the number of nonground non-monotonic rules of P. The number of atoms is limited by the Herbrand base. Therefore at most $x * c^v$ ground atoms can be propagated, where x is the number of different predicates in P. Therefore we get an upper bound for the prop function of $O((c^{v*a}) * v * r * x)$.

backtrack: The backtrack function has to remove each instance that has been added since the last positive guess. Removing an instance from the memory needs at most time v, where v is the number of variables occurring in P. There are at most $2 * x$ memories per rule storing instances. Therefore backtrack has an upper bound of $O(c^v * v * r * x * 2)$, since at most c^v instances can be in one memory.

Since we now have the runtime for all the base functions, this leaves us with the complete upper bound for the OMiGA algorithm of:

$$2^n * (n * (r * a * p + (c^{v*a}) * v * r * x + 1) + c^v * v * r * x * 2)$$

Obviously the propagation part dominates this formula, which thanks to O-notations leaves us with an upper bound of

$$2^n * n * c^{v*a} * v * r * x$$

where:

- x is the number of different predicates of P
- n is the number of ground non-monotonic rules
- c is the number of constants of P (the active domain)
- v is the number of variables in P
- a is the number of atoms in the rule with most atoms of P
- r is the number of nonground non-monotonic rules

Note that in terms of our input we have a nonground program P consisting of r nonground rules, each containing at most a atoms, as well as the active domain c, the predicates x and the variables v of P. The number of ground non-monotonic rules can at most be $c^v * r$, so we can replace n with $c^v * r$. Therefore in terms of input size we obtain an upper bound of:

$$2^{r*c^v} * r * c^v * c^{v*a} * v * r * x$$

We can reduce this to:

$$2^{r*c^v} * c^{v*(a+1)} * v * r^2 * x$$

While a double exponential runtime might not be terrific nothing else can be expected (if NEXP-Time $\neq$ EXP-Time) since OMiGA is capable of solving NEXP-Time hard problems.

CHAPTER 4

Representing ASP within a Rete network

In this chapter we introduce the reader to our approach of using a Rete network to calculate answer sets. For this purpose we will first have a look on how to represent an ASP program within a Rete network. Then we go into more detail and introduce the reader to the structure of OMiGA's Rete network.

4.1 A database-oriented view of Answer Set Programming

In order to improve the performance of our grounding on-the-fly answer set solver, we address the problem of determining all applicable ground rules for a nonground input rule by using a Rete network [7]. For this purpose we have a database-oriented look on the ASP input program and see every predicate as a table and each ground atom as one entry in the table for the corresponding predicate. Determining if there is an applicable ground rule is then simply the result of the join over all tables corresponding to the predicates within the rules body.

Example 4.1.1. *Let P be the following ASP program, stating that a node Y is reachable from a node X if Y is reachable from some node Z which is reachable from X within some graph G.*

$$reachable(X,Y) :- reachable(X,Z), reachable(Z,Y). \tag{4.1}$$

$$reachable(1,2). \quad reachable(2,3). \quad reachable(2,4). \tag{4.2}$$

Then from our point of view there would be only one table, namely for the predicate reachable, which has three entries namely (1,2), (2,3) and (3,4). To determine all applicable ground rule instances, we now join this table with itself over the variable Z, i.e. reachable(X,Z) $\bowtie$ reachable(Z,Y). (So the second position of an entry has to be equal to the first position of some entry *in order to be joined.) As a result this join returns (1,2,3) and (1,2,4). The ground rule instances corresponding to these joins would look like this:*

$$reachable(1,3) : -reachable(1,2), reachable(2,3). \tag{4.3}$$

$$reachable(1,4) : -reachable(1,2), reachable(2,4). \tag{4.4}$$

But rather than working with those rules, it suffices to add (1,3) and (1,4) into the reachable table when obtaining the join result (1,2,3) and (1,2,4).

A Rete network now is a graph like structure that is designed to optimize such join processes, by storing subjoins in order to avoid recalculation. For this purpose we create one node for each predicate of the input program. In order to determine a join for a whole rule, we now join nodes. For each two nodes that are joined an additional node is created which contains the result of the two joined nodes. Of course such joined nodes can be joined again, until the whole rule has been joined. Note that in our current implementation rules are joined in the order of their atoms, without applying any heuristics. Optimizing the join order is left for future work.

While this is the structure of the Rete network, its calculation of those joins is done iteratively. Whenever a new ground atom is derived it is added as an entry into the corresponding node. And then from this node only this new entry is pushed onward in order to be joined with the connected join partner, only creating potential new results that arise from this certain new instance. Of course all those results are again new entries in the joined node and are pushed forward as well until a leaf node is reached. If a leaf node is reached the arriving entries are new ground rule instances that can be applied.

Example 4.1.2. *Consider the following ASP rule:*

$$p(V,Y) : -q(V,W), s(W,X), t(X,Y). \tag{4.5}$$

Then we have three nodes, named after their corresponding predicate, for the rules body: q(V,W), s(W,X), t(X,Y), and one for its head: p(V,Y). Furthermore consider the following possible join order that is created by OMiGA:

1. *q(V,W) $\bowtie$ s(W,X) leading to a new node qs(V,W,X)*
2. *qs(V,W,X) $\bowtie$ t(X,Y) leading to a new node qst(V,W,X,Y)*

Now all ground facts of q,s,t are stored within the the corresponding nodes. The join nodes then contain the join between their children. Assume we have got the following facts: q(a,b), q(c,b), s(b,c), s(b,d), s(a,d), t(d,c). Then, the join node qs will contain (a,b,c) (c,b,c), (a,b,d), and (c,b,d). Therefore the join node qst will contain (a,b,d,c) and (c,b,d,c). Both of these instance yield a complete ground instances of (4.5)*, namely:*

$$p(a,c) : -q(a,b), s(b,d), t(d,c). \tag{4.6}$$

$$p(c,c) : -q(c,b), s(b,d), t(d,c). \tag{4.7}$$

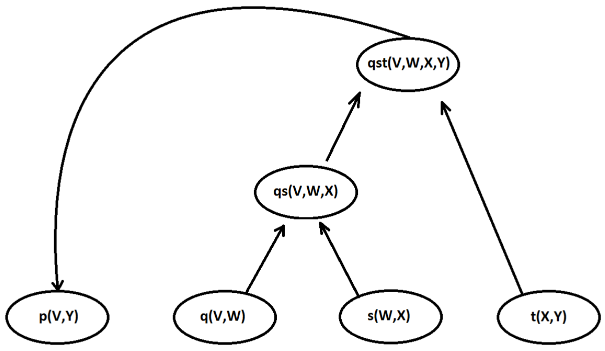

Figure 4.1: Representing the rule (4.5) within Rete

Therefore we can add p(a,c) and p(c,c) into the corresponding node p(V,Y).

Now let us assume that another ground atom t(d,d) is added to the Rete network. Of course this atom becomes an entry (d,d) in the t node. The benefit Rete now is that we do not need to recalculate the join for q and s since that result is already stored in the node qs. Therefore we can directly join the new entry with the qs node and obtain (a,c,d,d) and (c,b,d,d), which again represent complete ground instances of (4.5).

In a naive approach one had to recalculate the whole joins over and over again, whenever new facts are derived. With the Rete network we can avoid this, by storing the subjoins between two or more atoms in additional nodes. Intuitively speaking the more atoms within a rule's body, the longer the join order and the more beneficial is the use of a Rete network, since more subjoins are stored and therefore much more recalculation is avoided.

4.2 The Rete network of OMiGA

Note that although we could have used some already existing rule-based engine for our propagation phase, we decided to implement our own Rete network. The main reason for this decision was performance increase. Common rule based engines offer nice interfaces to use them with any data types. For example the Java rule-based engine DROOLS [30], can even store user defined classes within its Rete network. While this is convenient for a programmer, it creates overhead. Aside from that, Rete networks feature a negation that is different from the negation-as-failure which is needed for ASP. Furthermore a Rete networks supports deletion of facts when triggering rules, another feature not needed for ASP. In order to increase performance, we implemented our own Rete network that is closely bound to the actual data types we use within OMiGA to represent knowledge. Our results for programs that can be solved with propagation only (depicted in Figure 6.1) show that this approach is successful. As the Rete network is OMiGA's main component, the following section is dedicated to explain our Rete network in detail, by describing the structure and operations used. Implementation details concerning the storage and the various nodes of the join layer can be found in the implementation section (5.3).

For our Rete network we have two memories: The positive and the negative memory, where the positive one stores all predicate instances that were derived, and the negative one stores those instances that must not be derived anymore. Both memories operate upon three layers:

1. Basic layer
2. Selection layer
3. Join layer

While both memories are strictly separated for the first and second layer, the join layer is a mixture of both. The first layer serves as an entry point for new instances, while the second layer filters arriving instances for later use. Finally the third layer is the most complex one serving several purposes. It does all the joining, it evaluates operators, gathers instances of rules on which the choice unit can later guess and it instantiates the heads of rules. Each layer consists of nodes. Each node has a list of child nodes and a storage, that is used to store instances. The structure of our Rete network is depicted in Figure 4.2. A detailed description of how the storage works is given in Section 5.3.

Basic Layer The first layer of our Rete network solely consists of BasicNodes. For each predicate of the rewritten input program we have a BasicNode. BasicNodes correspond to exactly one predicate. Since predicates are unique, it follows that BasicNodes are unique as well. Their purpose is to store all instances that are derived for their predicate. When new instances arrive within a BasicNode, they are pushed to all children of that node. The children of a BasicNode can only be SelectionNodes. Note: For the negative memory NegativeBasicNodes are used. NegativeBasicNodes are the negative counterpart to normal BasicNodes, storing all instances for a predicate, that must not be derived anymore. Furthermore they feature closing which is discussed in Section 5.2.

Selection Layer The second layer of our Rete network consists of SelectionNodes. Each SelectionNode corresponds to one occurrence of a predicate within a rule. SelectionNodes do not store whole instances, but variable assignments corresponding to those instances. When we say a SelectionNode stores instances of a predicate, we actually mean it stores the corresponding variable assignment.

Example 4.2.1. *Consider the following example program:*

- *s(X,Y) :- p(X,Y), p(X,f(X)).*
- *t(X,Z) :- p(X,a), p(X,Z).*

Here we have four occurrences of predicate p within rules. p(X,Y) and p(X,Z) are of the same schema, as they only differ in Variable naming. Therefore only three SelectionNodes will be generated. One for p(X,Y)/p(X,Z), one for p(X,a), and one for p(X,f(X)). The first one will store all instances of p, because both terms are independent variables. The second one will store all instances of p, that have an 'a' at the second position, and the third one will store all *those instances where the second position is a function term called f and the first position of p equals that function terms first position. Note that there would be an additional SelectionNode if there was a predicate p(X,X), since the positions were not independent, this SelectionNode would not store all instances of p, but rather only those instances where first and second position are equal.*

SelectionNodes are children to BasicNodes. Whenever an instance is pushed from a BasicNode to a SelectionNode, the SelectionNode checks if that instances matches the node's schema. If so, the variable assignment is taken from the instance and stored within that SelectionNodes storage.

Example 4.2.2. *Let s be the selection node corresponding to the predicate p with schema (f(X), a), therefore corresponding to the atom p(f(X), a), and let i be the instance p(f(b),a) that is pushed into our Rete network. First i will be pushed to the BasicNode corresponding to the predicate p. The BasicNode will then push i to all its children. Since s corresponds to an occurrence of the predicate p, it is a child of that BasicNode and i will be pushed to s. The check for i matching the schema of s is positive since i contains a function term f at its first position and the constant a at its second position. The constant b within the function term will be taken as the variable assignment for X and stored within the storage of the SelectionNode s.*

The SelectionLayer is used in order to improve lookup performance. If there was no SelectionLayer, one had to join BasicNodes, which contain all instances of one predicate. Therefore one had to consider all the instances of a predicate and filter those that fit the atom that is currently considered. For example there might be the atom p(X,a) that should be joined with some other atom $q(X, Y)$. For this join we are not interested in all the instances of p, but only in those that have the constant a at the second position. Note that from a node n a join is not built only once but all the time when a new instance is derived that might be joinable with any instance contained in n. Thanks to our SelectionLayer we check instances once when they are derived and pushed to the selection layer. When building joins all instances are already stored in the corresponding SelectionNodes and therefore fit their join partners.

The children of a SelectionNode can be of any node type that occurs in the join layer. Note: For our negative memory we use only negative SelectionNodes, which serve the same purpose as normal SelectionNodes but are children of NegativeBasicNodes and feature closing: if a negative SelectionNode is closed, it will, when asked if a certain instance is contained, look into the corresponding positive SelectionNode's storage and answer with inverted truth value. Closing is discussed in more detail in Section 5.2.

Third Layer The main task of the third layer is to join variable assignments. Furthermore it evaluates operators, enforces constraints and derives new instances when rules are fulfilled. It also gathers guessable rules that are needed by the choice unit. For these different purposes several different types of nodes are used. The Join layer can contain the following types of nodes: join nodes, negative join nodes, operator nodes, choice nodes, head nodes and negative head nodes. A detailed description on how these nodes work can be found in Section 5.3.

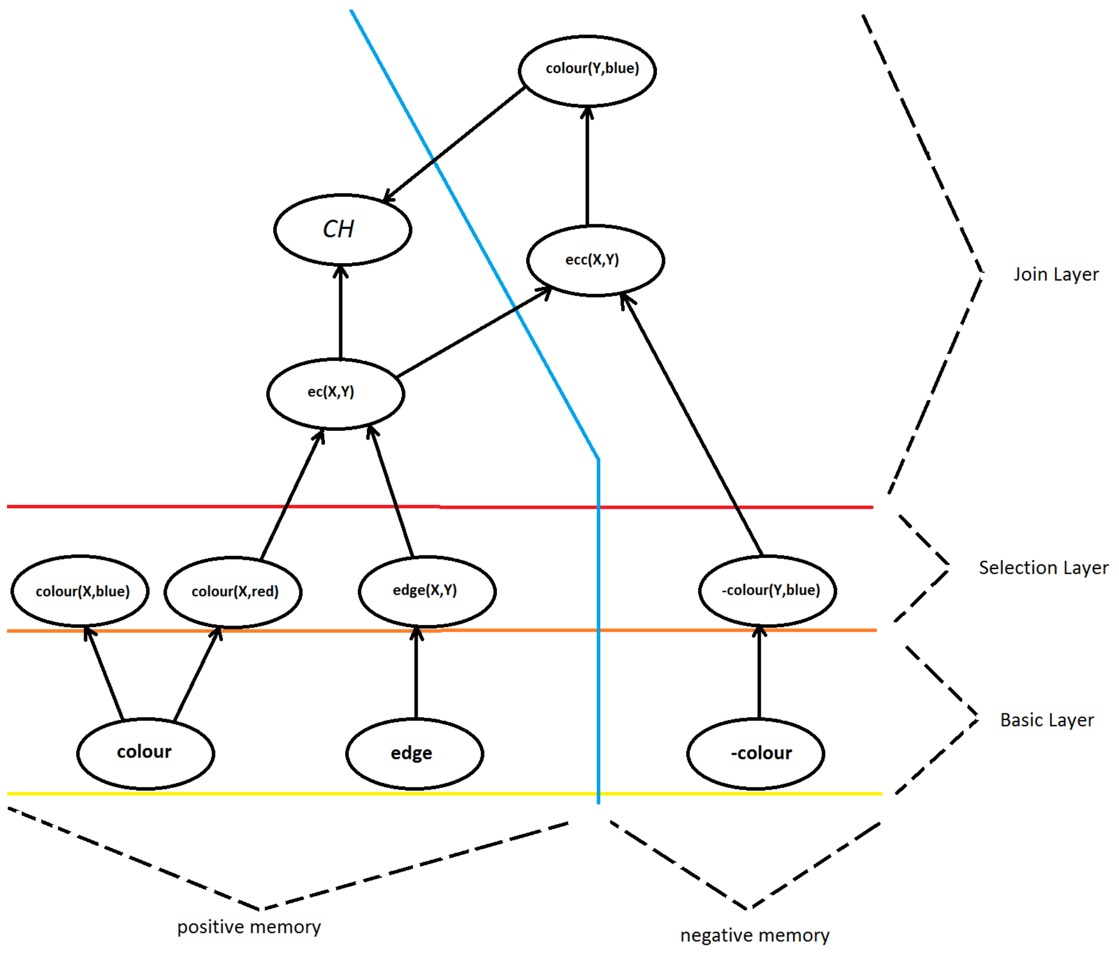

Figure 4.2: OMiGAs Rete network structured into memories and layers; nodes are represented by circles, arrows indicate childnodes (pushing of instances)

CHAPTER 5

System Architecture

This chapter is intended to give a deep insight into the OMiGA implementation. For this purpose we will first give a small overview of the whole system and how the different components work together. Then we will describe the function of the different components in detail in order to link the theoretical approach with the implementation. Finally we show which classes are used and how the most important parts are implemented.

5.1 Overview

The architecture of OMiGA can be divided into 4 main parts (depicted in Figure 5.1): the Parser, the Rewriter, the ReteBuilder and the OMiGA core, which in turn consists of the PropagationUnit, the ChoiceUnit and the Manager.

An OMiGA calculation starts with reading the input program with the parser component. For this purpose we have several Java entity classes to store the different ASP constructs: $Predicate$, $Atom$, $Term$, $FunctionTerm$, $Constant$, $Variable$, $Rule$. The parser takes the input and transforms it into our Java representation of the input program, which is then given to the rewriter.

The rewriter takes the Java representation of the input program and transforms it into an ASP program whose answer sets correspond one-to-one to the answer sets of the original program but whose non-monotonic rules have unique heads. This new program is then given to the ReteBuilder which creates a Rete network that corresponds to the rules of the rewritten program. It also pushes the facts of the input program into the Rete network, such that they will be triggered when the first propagation is called. After the Rete network is initialized the OMiGA core starts to work on this data structure. For this purpose the $Solve$ procedure (Algorithm 3.1 introduced in Chapter 3) of the Manager component is executed and the result printed to the console.

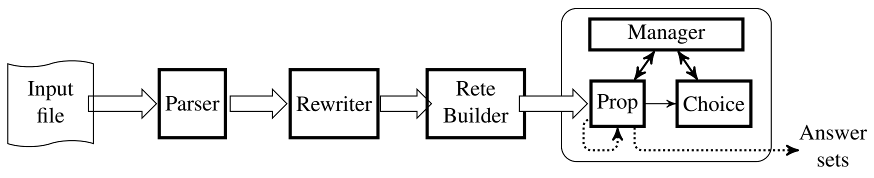

Figure 5.1: Boxes indicate entities; white arrows represent control and data flow during an OMiGA calculation; black arrows resemble control flow. The rounded box around the manager, choice, and prop node is referred to as the OMiGA core, which is in charge of the calculation

The OMiGA core consists of two working units (Propagation- and Choice-Unit) and one control unit (the Manager). The Manager alternates between calling the *prop* method of the propagation unit and the *choice* method of the choice unit until either a contradiction or an answer set is reached. Then the *backtrack* method of the choice unit is called to backtrack to a previous state in the calculation. Since both units work on the same Rete network there is no communication between the propagation unit and the choice unit. Nevertheless the changes done by the different operations have an influence on further calculation of the other unit. The Manager does not directly interact with the Rete network and is only capable of calling the functions: prop, inconsistent, choice and backtrack (As seen in Chapter 3.2.), where the first two are part of the propagation unit and the last two belong to the choice unit.

When all possible answer sets have been checked (i.e. the call of backtrack returns $\emptyset$) the Manager exits the main loop and the calculation terminates. Note that opposed to the theoretical algorithm given in Chapter 3.2, we actually do not gather all the answer sets and return them for printing, but rather we print them as soon as they are derived, since this is much more convenient.

5.2 Components

In this section we present the different components of OMiGA in the order they are used in an actual run depicted in Figure 5.1. We will give a detailed description of what those components are doing, but might not always talk about how they are doing it. This will be for example the case with the Rete networks nodes and the storage. In this case the information can be found in Section 5.3 - the implementation section, which offers more implementation relevant information.

Parser

The parser is written with the help of ANTLR (Another Tool for Language Recognition) [1] which is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks.

[1]http://www.antlr.org

OMiGA expects a standard ASP program as input without strong negation and without disjunction. OMiGA supports the following operators:

- relational operators: >,<,>=,<=,=
- algebraic operators: +,-,*,/
- assignment operator: IS

Note that we have an '=' and an 'IS' operator which more ore less do the same. While the '=' operator compares the left side to the right side the 'IS' operator only accepts one variable on its left side and assigns the calculated value of the right side to it. The difference is that when using the assignment operator new constants can be created.

Example 5.2.1. *Consider the following rule:*

$$h(X, Y) : -b(X), Y\ IS\ X + 2.$$

Here a new constant could be created. For example if b(5) is derived and there is no constant 7 at the moment this constant will be created as soon as the rule fires, since h(5,7) is derived. This could not be achieved with a standard '=' operator since:

$$h(X, Y) : -b(X), Y = X + 2.$$

would not be safe, by our definition.

The reason for having two such operators is simple: As long as one only uses the '=' operator (and no function terms) one will only deal with a bounded domain, while when using the IS operator one might run into an endless calculation because an infinite domain is created.

Function terms

OMiGA supports the use of function terms of arbitrary nesting depth. Thanks to grounding on the fly OMiGA has no problems arising from an initial grounding of infinitely many function terms. Nevertheless when using function terms one might run into the danger of creating an infinite active domain.

Example 5.2.2. *Let us consider the following rule:*

$$pred(f(f(X))) : -pred(f(X)).$$

Obviously the calculation of this rule can not terminate as soon as one instance of pred(f(X)) has been derived, since then pred(f(f(X))) is derived and the new function term f(f(X)) is created. But from this the rule triggers again leading to pred(f(f(f(X)))) while creating the new function term f(f(f(X))) and so on.

So while OMiGA has absolutely no restrictions on the use of function terms, the user has to pay caution to not create infinite domains which will lead to endless calculation and probably to no output.

Rewriter

The main goal of the rewriting is to create an answer set equivalent program, where all non-monotonic rules have unique heads. This enables the OMiGA engine to know from which rule a certain guess came, without using any further data structures. Thus when guessing a rule not to fire it is sufficient for the OMiGA engine to just put the head of that ground rule instance into the negative memory. Without this rewriting we would need to keep track of such rules and integrate another check if such a forbidden rule is firing. Therefore we decided to use the rewriting in order to solve all this by our Rete network alone.

Another goal of the rewriting is to add further rules that shift workload from the choice unit to the propagation unit. These rules are created by analyzing the structure of the input rules and restructuring them.

Unique Heads

In order to ensure unique heads we parse each input rule r and replace it by the following two rules:

1. $newHead : -body(r).$
2. $head(r) : -newHead.$

where $newHead$ is an atom named $rule\_rID\_pred(v_1, ..., v_n)$, $pred$ is the name of the predicate of $head(r)$, rID is a unique id for that rule, and $v_1, ..., v_2$ are all the variables contained within $body(r)$. Since $newHead$ contains rIDwithin it's head it is unique within the rewritten program.

Intuitively we only inserted a step between the derivation of the old head and the old body, since it still holds that $head(r)$ is derived whenever the $body(r)$ holds. Therefore nothing has changed concerning the answer sets of the program, except for the newly introduced predicates that can be projected away in order to obtain the answer sets for the original program. Facts and constraints are not rewritten and taken as they are.

Negative Rules

For the purpose of optimization we create some additional negative rules. Negative rules resemble normal rules, but when fulfilled they add their head to the negative memory, rather than into the positive memory. Note that the negation used for these negative rules is not default negation. For example:

$$\neg a : -b_1, ..., b_l, \neg c_1, ..., \neg c_m.$$

Such rules are not supported by our innate language in the current version and can only come from this rewriting. For each rule of the input program we have a look at each atom a of its body. If a contains all variables that are contained in $head(r)$ then we create the following rule:

$$\neg head(r) : -\neg a.$$

Example 5.2.3. *So for example from:*

$$p(X) : -q(X), not\, s(X).$$

we obtain:

$$\neg p(X) : -\neg q(X).$$
$$\neg p(X) : -s(X).$$

These rules help to detect ground rule instances that won't fire anymore. Another negative rule is created in order to shift the knowledge of a new head atom not being derivable anymore to the original head atom. This rule contains all rewritten heads of the positive rewritten rules as negative body atoms and the original head atom as negative head.

Example 5.2.4. *So for example if there are two rules:*

1. $p(X) : -q(X).$
2. $p(X) : -s(X).$

That were rewritten because of our unique head rewriting resulting in:

1. $p\_rule1(X) : -q(X).$
2. $p\_rule2(X) : -s(X).$
3. $p(X) : -p\_rule1(X).$
4. $p(X) : -p\_rule2(X).$

Then the following negative rule would be created as well:

$$\neg p(X) : -\neg p\_rule1(X), \neg p\_rule2(X).$$

Example 5.2.5. *To state a complete rewriting example consider a program consisting of the following rules:*

$$\begin{aligned} pred2(X) &: -dom(X,Y), not\, pred1(X,Y). && (5.1)\\ pred1(X,Y) &: -dom(X,Y), not\, pred2(X). && (5.2)\\ pred1(X,Y) &: -dom(X,Y), not\, pred3(X,Y). && (5.3) \end{aligned}$$

these are rewritten to:

$$\mathit{rule1\_pred2}(X,Y) :- \mathit{dom}(X,Y), \mathit{not}\ \mathit{pred1}(X,Y). \tag{5.4}$$
$$\mathit{pred2}(X) :- \mathit{rule1\_pred2}(X,Y). \tag{5.5}$$
$$\mathit{rule2\_pred1}(X,Y) :- \mathit{dom}(X,Y), \mathit{not}\ \mathit{pred2}(X). \tag{5.6}$$
$$\mathit{pred1}(X,Y) :- \mathit{rule2\_pred1}(X,Y). \tag{5.7}$$
$$\mathit{rule3\_pred1}(X,Y) :- \mathit{dom}(X,Y), \mathit{not}\ \mathit{pred3}(X,Y). \tag{5.8}$$
$$\mathit{pred1}(X,Y) :- \mathit{rule3\_pred1}(X,Y). \tag{5.9}$$
$$\neg \mathit{pred2}(X) :- \neg \mathit{rule1\_pred2}(X). \tag{5.10}$$
$$\neg \mathit{rule2\_pred1}(X,Y) :- \neg \mathit{dom}(X,Y). \tag{5.11}$$
$$\neg \mathit{rule3\_pred1}(X,Y) :- \neg \mathit{dom}(X,Y). \tag{5.12}$$
$$\neg \mathit{rule3\_pred1}(X,Y) :- \mathit{pred3}(X,Y). \tag{5.13}$$
$$\neg \mathit{pred1}(X,Y) :- \neg \mathit{rule2\_pred1}(X,Y), \neg \mathit{rule3\_pred1}(X,Y). \tag{5.14}$$
$$\neg \mathit{pred2}(X,Y) :- \neg \mathit{rule1\_pred2}(X). \tag{5.15}$$

Rules 4.4 to 4.9 derive from the unique head rewriting, rules 4.10 to 4.13 are negative rules to help closing ground instances of predicates obtained by the rewriting, and rules 4.14 and 4.15 push the knowledge of unsatisfiable ground instances to the corresponding original head predicates.

Note that the main focus of our work was the grounding on the fly evaluation using the Rete network, therefore the current rewriting was focused on making non-monotonic rule heads unique and is not optimized. In the current implementation, an overhead of rules is created, slowing down the calculation. Because of this we advise to deactivate rewriting for positive input programs as well as for programs where non-monotonic rules are already unique. Also we think that there is much potential within the rewriting component of OMiGA and further analyzing the input rules might lead to some more efficient rewriting.

The Rete Builder

To build the Rete network the class *ReteBuilder* is used. It takes an ASP program as input and builds the network rule by rule. In order to understand how the ReteBuilder works, one must first understand the basics of our Rete implementation which was presented in Chapter 4.

The Rete network is built rule by rule. For each rule first all atoms of the positive body of a rule are used from left to right. BasicNodes and SelectionNodes are built from them (if not already there, due to other rules). Then the first and second SelectionNode build a joinNode, that becomes child to both. Then we continue with this JoinNode and the SelectionNode corresponding to next positive atom to obtain a new JoinNode and so on, until there are no further positive atoms left. Then all operators are applied. We start with the first operator from left and add a corresponding OperatorNode as child to the last JoinNode. The next OperatorNode is added as child to that OperatorNode and so on until no more operators are left. Then we continue with the current node and the first negative atom to build a NegativeJoinNode. The SelectionNode corresponding to next negative atom will be joined with that NegativeJoinNode. We continue like this till there are no more negative atoms left. Then we put a HeadNode containing the head of the rule on top of the current node. Furthermore we add a ChoiceNode as a child to the left child of the first NegativeJoinNode and the HeadNode. Negative rules are built the same way, except they have a NegativeHeadNode instead of a normal HeadNode.

Example 5.2.6. *Figure 4.2 shows the Rete network representing the following rule.*

$$colour(Y, blue) : -colour(X, red), edge(X, Y), not\ colour(Y, blue). \tag{5.16}$$

All nodes within Figure 4.2 are built because of this rule, except for colour(X,blue), which is there for the only purpose to show that BasicNodes can have several children. First the BasicNodes colour, edge and -colour are built. Then the SelectionNodes colour(X,red), edge(X,Y), -colour(Y,blue) are built and added as children to the corresponding BasicNodes. Now the ReteBuilder starts with the positive body of the rule and therefore joins colour(X,red) with edge(X,Y) and creates the JoinNode ec(X,Y). Note that the constant 'red' is dropped within the ec node, since we only store variable assignments, and because of the Rete structure any variable assignment within the ec node corresponds to exactly one partial ground instance of the rule, for which vertex X of edge(X,Y) is red. Since there are no further atoms in the positive body of the rule, and no operators are contained, the ReteBuilder now joins the ec node with -colour(Y,blue) to create the negative JoinNode ecc(X,Y). Again the constant 'blue' is dropped. One may notice, that the size of variable assignments stored within the ecc node did not increase in comparison to the ec node. Since we only deal with safe rules, all variables of the negative rule are within the positive body, therefore negative JoinNodes never increase the size of the variable assignments stored. Since there are no further atoms in $body^-(r)$ the ReteBuilder puts a HeadNode on top of the ecc(X,Y). Finally a ChoiceNode is added as a child to that head node and the ec(X,Y) node, because it was the last positive node within this join order.

Propagation Unit

The Propagation-Unit heavily depends on the Rete network, and actually is integrated into it. Since all needed data structures have already been described in Section 4, all that is left to show here is how an actual propagation step is handled:

- Whenever a new instance is pushed as a fact into the Rete network, this instance is stored within the BasicNode of the corresponding predicate and marked as not yet propagated.
- When propagation is called, all BasicNodes are asked to push their marked instances to all their children, and unmark them. This is done sequential one instance at a time.
- all other nodes will do some calculation with the arriving instances and in case a new instance is stored within a node, this instance is pushed to all its children as well. Headnodes will add new instances into BasicNodes and mark them.

- the propagation finishes when no BasicNode has any marked instance left (including those that were derived during this propagation by HeadNodes)
- the propagation aborts if inconsistency is detected

To start a propagation one can simply call the method propagate() of the Rete network.

Example 5.2.7. *Consider the following program*

$$connected(X,Z) : -connected(X,Y), connected(Y,Z). \tag{5.17}$$

$$connected(1,2).connected(3,4).connected(2,3). \tag{5.18}$$

This is a small example with only one predicate. The program finds pairs of vertecies that are connected. The Rete Network in this case consists of one BasicNode for the predicate connected and one SelectionNode for the predicates occuring schema with a variable at each position. One JoinNode with left and right parent both being the SelectionNode, and a HeadNode connected(X,Z). At startup all three facts are added into the connected-BasicNode. Then when propagation is started connected(1,2) is chosen and pushed to all children, in this case the only SelectionNode. (All facts match the SelectionNodes schema, since both positions can contain anything). There are no partners for a join, since the only instance within the SelectionNode at the moment is (1,2) which does not match the criterion. So the next fact connected(3,4) is pushed into the selectionNode. Again no join partners are found, since neither (1,2) nor (3,4) are a match for (3,4). So the third fact connected(2,3) gets pushed into the SelectionNode. (2,3) is being pushed form left to the JoinNode, where one partner is found (3,4). A new variable assignment (2,3,4) is added into the JoinNode and pushed to the HeadNode, which leads to the generation of a new fact: connected(2,4). This fact is added into the BasicNode and marked. But (2,3) is also pushed into the JoinNode from the right, finding one join partner (1,2) leading to (1,2,3) being added into the JoinNode. Which is also pushed into the HeadNode leading to the new fact: connected(1,3), which is added into the BasicNode as well. Now the fact connected(2,4) is pushed from the BasicNode into the SelectionNode from left but no join partners are found. From right one is found, namely (1,2) leading to (1,2,4) being added into the JoinNode and then pushed to the HeadNode leading to the generation of connected(1,4), which again is added into the BasicNode and marked. Next connected(1,3) is pushed into the SelectionNode. One join partner is found when pushed from the left, namely (3,4) leading to (1,3,4) being added to the JoinNode and then being pushed to the HeadNode. No new fact is added into the BasicNode because connected(1,4) is already within the BasicNode. Finally connected(1,4) is pushed to the SelectionNode and no join partners are found. No further facts are left to propagate, a fixpoint is reached and the propagation stops. For this positive example program the calculation would be over and the found answer set would be: (1,2) (1,3) (1,4) (2,3) (2,4), (3,4).

Choice Unit

The choice unit is responsible for the non-monotonic part of the input program and therefore features three main functions: guessing, closing and backtracking

Guessing

When no more propagation can be done, the solver has to guess. In order to determine the ground rules where we can do a guess, the choice unit has access to the ChoiceNodes of our Rete Network, which store exactly those ground rules. When guess() is called one such instance is taken from one of the choice nodes and either a positive or a negative guess is made. If a positive guess is made we unify all negative atoms of the corresponding rule, and add them as instances into the negative memory. Therefore the next call to propagate will add the head of that rule into the positive memory, since the rule is then supported and unblocked. If a negative guess is made for a rule r then the grounded head of r is added into the negative memory. Therefore inconsistency will be detected whenever that rule fires. Note that this is correct only because all heads of rules we guess on are unique thanks to the rewriting.

A negative guess will be made, if the last operation before the guess was a backtrack. Otherwise a positive guess will be made. (So the first interpretation checked will be that one with all guessable ground rules applied, and the last one will be the one with all those rules guessed to be not applied) If there is no more ChoiceNode containing an instance, no more choices can be done and the calculation has finished, meaning we reached an answer set if no inconsistency was detected.

In order to optimize the choice process, we use a dependency graph in order to determine from which ChoiceNode we should pick the next ground instance to choose on. The idea behind this is to guess in such a way, that predicates on which other predicates depend, are calculated first.

Example 5.2.8. *Consider the following rules:*

$$p(X,Y) : -s(X), t(Y), not\, u(X,Y). \tag{5.19}$$

$$q(X,Y) : -s(X), k(Y), not\, p(X,Y). \tag{5.20}$$

Then it is wiser to first guess on the first rule, since this will derive new instances for p(X,Y), which will reduce the number of guesses we need to do for the second rule. If we guessed on the second rule first, this would have no influence on the number of guesses for the first rule, since only the second rule is dependent to the first but not visa versa.

To achieve this we build a dependency graph, where each predicate is represented by a node. A directed edge is created from a node being within the body of a rule, to the head of the rule. Constraints are ignored. From this graph the strongly connected components (SCCs) are calculated and ordered in such a way that SCCs depending on other SCCs are placed after those. The whole SCCs calculation and ordering is done by an external Java library called jgrapht [2], which returns a list of SCCs ordered as described above.. From this ordered list of SCCs we determine which of them contain a predicate that is head of a non-monotonic rule. When a guess is done, ChoiceNodes of rules of lower SCCs are picked first. Note that when all ChoiceNodes of an SCC are empty that SCC will be closed (see section 5.2).

Please note that apart from this, OMiGA has no intelligent heuristics concerning the order in which guess are made. OMiGA first goes through all instances of the first ChoiceNode of an

[2]http://jgrapht.org

SCC before going to another ChoiceNode within the same SCC. Here we see great potential for further optimization.

Closing

Another important benefit of SCCs is closing. Whenever no more guesses are available within the lowest non closed SCC, we can close it. This means no fact for predicates contained within this SCC can be derived anymore. When we close an SCC, we close all NegativeBasicNodes of predicates within that SCC. Those BasicNodes will, from this time on, rather look into the storage of the corresponding positive BasicNode, and answer with the inverted truth value for membership. All selectionNodes that are children to those basic Nodes are closed as well. NegativeSelectonNodes that are closed will look into the storage of the corresponding positive SelectionNode and answer membership requests with inversed truth values. For those closed predicates everything that is not derived is seen as out. But in order to have a correct closing, we have to recalculate affected NegativeJoinNodes, since now a bunch of facts might have to be seen as out (but did not trigger a push). Therefore when clsoing a SelectionNode it also informs all its children. If a negative JoinNode gets informed that its right parent is closing, for all instances of its left parent, the now closed partner is asked for membership. If this yields true (meaning that instance is not derived within the positive memory), the instance is stored within the negativeJoinNode and pushed to all children.

Closing is a powerful tool, since we do not have to put all facts that are out into the negative memory, but rather it lets us define a closed world assumption such that all facts of a given predicate that are not within the positive memory are out, without using any additional space.

Backtracking

Backtrack is a function restoring our Rete network to the state it was in before the last positive guess. In order to realize this we use the following data structures to incrementally store the state of our Rete network:

```
ArrayList<LinkedList<Pair<Node,Instance>>>
decisionLayer
```

This structure is used for every node type. The position within the array list defines the decision layer. For each decision layer we store all instances and the nodes they were added to, within a list. Instances are stored into this structure when they are saved to a nodes storage within Rete. When backtrack is called we take all instances of the last decision layer, and for each of them we tell the corresponding node to delete that instance.

While the first structure saves all arriving instances of all nodes, the following structure keeps track of the instances that are deleted from nodes at a certain decision layer.

```
ArrayList<HashMap<ChoiceNode,HashSet<Instance>>>
choiceNodesDecisionLayer
```

ChoiceNodes are the only nodes where instances can be removed aside from backtracking. This happens if a HeadNode is reached that links back to a ChoiceNode. The corresponding instance is deleted from the ChoiceNode, since no more guess is needed for this rule instance. When backtrack is called, all instances that are stored here for the last decision layer are added again into the corresponding ChoiceNodes. Like within the first structure the position in the ArrayList determines the decision layer. For each decision layer we store the choice nodes, where instances have been removed, as Keys of a HashMap. For each choice node, the removed instances are stored within a separate HashSet.

The Manager

While the propagation and choice unit offer the main methods to calculate the answer sets of an input program, the Manager is needed to coordinate the whole process. After the program has been read and rewritten and the Rete network has been initialized, the Manager's $Solve$ procedure is called which controls the depth-first search for answer sets as follows:

It initializes the SCC data structure by calling *DeriveSCC()* of the choice unit. Afterwards the methods *propagate* (propagation unit) and *choice* (choice unit) are called in alternation. If inconsistency is raised or no more choices are possible, backtrack() is called. In case of no more choices an answer set is printed. If no more backtracking can be done, the Manager terminates and has found all answer sets.

An algorithmic description of the Manager is given in Figure 3.1. Note that we implemented OMiGA in such a way that the calculation process can be controlled easily, which is achieved by offering simple methods for:

- propagation
- choice
- backtrack

This way one has full control over OMiGA by calling just those three methods, without the need of knowing which data structures are used within OMiGA. The major benefit arising from this approach is that one can calculate answer sets stepwise. This enables one to write a new Manager component that for example can handle several ASP contexts combining them into an distributed ASP system, without adapting the rest of OMiGA. Another interesting adaption of the Manager component might be to first let OMiGA calculate to some point and then add further facts by user input or from another program.

5.3 Implementation

In the following we will focus on the most important parts of the OMiGA implementation. For this purpose we will first repeat some knowledge of hash-maps and -sets, since OMiGAs data structures are heavily based on them. Then we present our entity classes. Entity classes are used to represent ASP programs and therefore are also used during a calculation for matching or generating new stuff. Finally we present the two main reasons behind OMiGAs calculation speed: The storage and the Rete network. Note that we have already dedicated several section to the Rete network, but now we will go into detail about how the different nodes are working.

Hash Maps, Hash Sets and Uniqueness

This subsection contains information that is crucial to understand the performance behind OMiGAs data structures therefore it is repeated here in detail.

A hash map is a set of key-value pairs. In order to find a value within a hash map one has to state the key and the hash map returns the corresponding value in constant amortized runtime O(1). For this to work, objects that are used as keys within a hash map have to implement the *hashcode()* and *equals()* function. The hash code is used to identify a bucket within the hash map. If there is only one entry for this hash code the corresponding value is immediately returned. If there are more keys with the same hash code, those keys are distinguished by using the equals method. A hash set is the same, but each value is its own key. Since OMiGA's calculation builds on continuously adding and looking up values within such hash structures, it is important to implement a fast *hashcode()* and *equals()* function. Therefore all our entity objects implement these methods like this: for the hash code we simply calculate an integer value when creating a new object. That integer value is then always returned when *hascode()* is called. For the equals method, we use the equality (==) operator of Java. This operator compares the memory addresses of two objects and therefore is pretty fast. But of course, in order to use this, we have to ensure uniqueness of entity-class objects that use the equality operator. This is realized via a factory pattern. Our entity classes can only be created via a static factory method. For example to create a new Predicate rather then writing *new Predicate(name, arity)*, write *Predicate.getPredicate(name, arity)*. The factory pattern works like this: each class has a static data structure for storing its instantiated objects. If an object is requested via the factory method, the object that matches the newly requested item is returned if already existent. Only if there is no such object a new one is created and returned. This way all objects occurring during a calculation are unique. If we did not ensure uniqueness, there could be for example two constant objects c1, c2 that both encapsulate the variable c. Then a comparison like c1==c2 may yield false, although both constants represent c.

Entity classes

In order to represent an ASP-Program, OMiGA uses the following entity classes: Predicate, Atom, Constant, Variable, FunctionTerm, Instance. These classes are optimized for use within hash maps and hash sets on which OMiGA heavily relies on, for reasons of performance.

Predicate

```
class Predicate {
    private String name;
    private int arity;
    private int hashcode;
}
```

A predicate is an entity object resembling a logical predicate of ASP. It encapsulates a name and an arity. Note that a predicate only equals another predicate if both the name and the arity are equal. Predicates of same name but not same arity are considered to be different predicates.

Atom

```
class Atom {
    private Predicate p;
    private Term[] terms;
    private int hashcode;
}
```

An atom is an entity object representing a logical atom of ASP. It encapsulates a predicate and a list of terms. An atom is an occurrence of a predicate within a rule. The terms of an atom specify a certain schema determining which predicate instances may be used for this atom when grounding. So for example $p(X, a)$, specifying that only ground instances of p might be used for this atom, where a is at the second position. There can be multiple atoms for the same predicate, but they have to differ in terms.

Term

```
class Term {
    protected String name;
    protected int hashcode;
}
```

A term is either a Constant, Variable or FunctionTerm and the basic object to build atoms and instances. The name variable is the name of the corresponding logical object this term represents. The hash code variable stores the hash value for this object, which is calculated once, at object creation.

Constant

```
class Constant extends Term {
    private Integer intValue;
}
```

A constant is an object that represents an ASP constant. It encapsulates a single String in our ASP domain, that is stored in the name variable. For example a, $paul$ or 1. For constants representing integers we parse the constant name and store it within the intValue. This fastens up later use of operators like X = Y + Z.

Variable

```
class Variable extends Term {
    private Term value;
}
```

A Variable is an entity object that represents an ASP variable. It has a name and can have a constant or function term as value, which is used during calculation. The value variable is used for calculation within the Rete network. Here, when matching instances and creating variable assignments, the involved variables values are set to the corresponding instance value. The resulting variable assignment is the obtained by taking all values of the desired Variables.

FunctionTerm

```
class FunctionTerm extends
Term {
    private ArrayList<Term>
children;
}
```

A function term is an entity object representing an ASP function term. It is defined by it's encapsulated name and list of terms: for example $f(X, a, g(Y))$.

Instance

```
class Instance {
    private Term[] terms;
}
```

An instance encapsulates an array of terms. During a calculation it is used to represent variable assignments, and in combination with a predicate represent facts. They are stored within the nodes of OMiGA's Rete network. If for example an instance $[a, b, c]$ is stored within a node representing the predicate p of arity 3, then this represents the fact $p(a, b, c)$. In combination with a whole rule an instance represents a ground instance of that rule. And within the join nodes of our Rete network, they represent partial ground rule instances.

Storage

Each node in our Rete network has its own storage where all the variable assignments are stored as instances. For this purpose we have the storage class that is a wrapper for an array of hash maps with terms as keys and hash sets of instances as values, or in Java code:

```
HashMap<Term,HashSet<Instance>>[]
```

The meaning of this data structure is the following: each position of this array of hash maps resembles a position of the variable assignment, sorted by terms. So an instance $[a, b, c]$ will be indexed three times, once for the first position under a, once for the second position under b and finally at the third position under c.

In order to query the storage one has to specify a selection criterion. The selection criterion is an array of terms of the size of this storages hash map array (so in the size of the variable assignments that are stored within this storage). A constant or grounded function term within the selection criterion specifies that for this query we want instances where that position is equal to the specified constant (resp function term), while a variable is used as a wildcard. The storage class now offers two methods to retrieve information about the stored instances.

- The contains method needs a selection criterion where no variables are allowed. It returns true if the instance corresponding to the selection criterion is contained within this storage and false otherwise. This is done in amortized linear time.
- The selection method returns all instances stored within this storage that match the selection criterion. For this purpose all positions that do not contain a variable in the selection criterion have to be checked resulting in a runtime of $O(k * s)$, where k is the number of instances contained for the term with least instances stored and s is the number of positions with no variable in the selection criterion. In case that all but one variable are set in the selection criterion, the runtime is reduced to constant time $O(1)$, since we simply return the hash set for that position and term.

While we do index instances multiple times with this data structure we obtain good runtime for the operations that are most important for our Rete network. The nodes of our Rete network call both the contains and the selection method of the storage class every time they have to do a join. The selection method is hereby used within the positive memory while, because of our safety criterion, the faster contains method suffices for the lookups needed in the negative memory.

Rete network

An overview over our layered memory structure has already been given in Section 5.2, therefore we will now only complement those parts that need a more complex analysis. For this purpose we analyze all the nodes of the join layer in detail and give examples on how an actual join between two nodes works.

JoinNodes

A join node is the child of two other nodes that can be either: JoinNodes, SelectionNodes or OperatorNodes. They represent the join between their two parents, and store all variable assignments, that occur from this join. Every time a new instance arrives in one parent, the JoinNode asks the other parent for all contained instances that can be joined with the arriving instance. If there are joinable instances, the new instance builds new variable assignments with each of them, which are then stored within the JoinNode and pushed to all its children. In order to achieve this, each JoinNode contains three arrays that are used in order to ask for the needed instances and to create the new variable assignment.

```
Integer[] selectionCriterion1
Integer[] selectionCriterion2
int[] instanceOrdering
```

As described in the storage section (5.3), in order to get instances from a node the selection method is used. The selection method needs a selection criterion that determines which instances shall be returned. The purpose of the two arrays *selectionCriterion1* and *selectionCriterion2* is to determine the selection criterion for the newly arrived instance in a performant manner. The $instanceOrdering$ array is used to specify how the new variable assignments that arise from this join shall look like.

The selectionCriterion arrays are of the size of the corresponding parent's stored instances. Therefore $selectionCriterion1$ belongs to the left parent while $selectionCriterion2$ belongs to the right one. The array positions correspond to the positions of the selection criterion that will be created. The numbers within the arrays resemble the position of the other join partners instances. So if there is a number x at position i within $selectionCriterion1$ then this means that in order to build the selection criterion, that will be sent to the left partner, will contain at position i the term that is located at position x of the arriving instance of the right partner.

The first position of an instance is actually referenced by 0, the second by 1, and so on. Some positions of the selectionCriterion arrays may not contain numbers, but null. If there is null at position i, then there will be a variable at position i of the created selection criterion (meaning a wildcard at that position for selecting instances from the join partner).

Example 5.3.1. *So for example let p(X,Y) be the left partner and q(Y,Z) be the right partner of a join. Then the two partner arrays look like this:*

- *selectionCriterion1:[null,0]*
- *selectionCriterion2:[1,null]*

selectionCriterion1 states that for the left partner the first position might contain anything while the second position has to match the content of the first position of the arrived instance of the right partner. selectionCriterion2 states that for the right partner the first position must match the second position of the left partner and the second position might contain anything. Whenever an instance arrives at one of the partners, a selection citerion is built from the array of the other partner and that arrived instance. So for example q(a,b) arrives within the right partner, then a selection criterion [X,a] will be obtained which can then be used to look up the instances of the left partner.

The benefit of these arrays is that the desired selection criterion can be built directly, because the JoinNode knows exactly which variable belongs to which position. Otherwise the JoinNode would have to ask the parents for their variable positions and match them to achieve this, which would cost time.

When all the join partners are gathered the *instanceOrdering* array is used to create a new variable assignment that is then added into the JoinNode. The positions of this array correspond one-to-one with the positions of the instance that will be added into the JoinNode. The numbers stored within this array define how the term of that instance's position is derived. A positive number indicates the right partner's instance is needed, while a negative number indicates the left partner's instance. The number 1 stands for the first position, 2 for the second and so on.

Example 5.3.2. *So in our example the instanceOrdering array would look like this: [-1,-2,2]. Meaning that for the first position of our new variable assignment we have to take the first position of the left partner, for the second one the second position of the left partner and for the third position the second position of the right partner.*

Again this enables a JoindNode to directly build the variable assignment, rather than comparing Variable positions of parents.

NegativeJoinNodes Joins between the positive and negative memory are treated with special negative JoinNodes. The right partner of a NegativeJoinNode will always be a negative SelectionNode. For NegativeJoinNodes the joining differs depending from which partner the instances arises. If it is the negative one, we have to do the standard calculation we do for normal JoinNodes as well. But if the instance arrives from the positive partner, we simply ask the negative node for containment of that instance, and, if it is contained, add it. Therefore NegativeJoinNodes are more performant than normal ones. Note that this can be done, since we only deal with safe rules, meaning that all variables of the right partner are already fixed by an assignment of the positive partner. Furthermore NegativeJoinNodes feature closing, which is described in Section 5.2.

Example 5.3.3. *Let qw(X,Y) be a NegativeJoinNode that joins a node q(X,Y) from the positive memory with the NegativeSelectionNode w(Y). If an instance [b] arrives in w(Y), qw(X,Y) has to ask q(X,Y) to return all instances with b at it's second position (by using the selection method of the storage class). But if an instance [a,b] arrives in q(X,Y), qw(X,Y) only has to ask w(Y) for membership of instance [b] (which is efficiently done by calling the contains method of the storage class).*

HeadNodes

HeadNodes represent the head of a rule and therefore are stored at the end of a complete join as a leaf node. They contain the head atom of the corresponding rule, and whenever an instance is pushed into a head node, rather than storing it, the Headnode grounds the head atom according to the arrived variable assignment. Then the grounded head atom is pushed as an instance into the corresponding BasicNode. If the corresponding rule is a constraint, the HeadNode signals that the current partial interpretation is contradictory, and therefore stops further propagation.

Negative HeadNodes NegativeHeadNodes do the same as standard Head-Nodes but add to the negative memory. Because our input language currently does not support such negation, these nodes can only occur from our internal rewriting.

ChoiceNodes

ChoiceNodes are special nodes that occur as children of the last positive node of a rule that also has a negative body. All instances of the last positive node are also pushed into the ChoiceNode. A ChoiceNode furthermore is child to the HeadNode of the corresponding rule, and each instance that reaches the HeadNode is deleted from the ChoiceNode. This way a ChoiceNode contains exactly those instances of the corresponding rule that are supported, not blocked and not unblocked. So we obtain exactly those instances that we want to guess on.

OperatorNodes

OperatorNodes represent relational operators or the assignment operators. In case of relational operators like X = Y, the OperatorNode functions like a more elaborate SelectionNode, by simply checking if the position that represents X matches that of Y. For more complex operators like V + W = X + Y the equation is solved by first replacing the variables by the values of the current instance and then using the standard mathematical operators on them. For the assignment operator, OperatorNodes only calculate the right side of the equation that way, and extend the arriving instance by another position representing the value for the variable of the left side of the operator.

5.4 Usage

OMiGA is implemented in Java and comes as a .jar file. It can be executed on the command line as follows:

java -jar omiga.jar <filename> [-answersets=X, -filter=predicateName -rewriting=[1|2]

The $filename$ parameter is mandatory and corresponds to the path of the file containing the input program. If no further parameters are specified all answer sets will be computed and answer sets will be printed to the console. The $answersets$ parameter defines the number of answer sets OMiGA computes. The calculation terminates either after X answer sets have been found or, if there are less answer sets than X for the input program, after all answer sets have been derived. Note that for $X = 0$ a computation over all answer set will be triggered. So this has the same effect as calling OMiGA without the $answersets$ parameter. With the $filter$ predicate one can specify predicates that should be printed. All other predicates are not printed when printing answer sets. The $rewriting$ parameter indicates whether a calculation with or without rewriting is started, where 1 indicates that the input program is rewritten and 2 indicates it is not. Note that when omitting this parameter OMiGA assumes 1 as default and rewrites the input program. Note that a calculation without rewriting can lead to wrong results if the input program contains non-monotonic rules that have no unique head within the input program.

CHAPTER 6

Evaluation

In this chapter we compare OMiGA to state-of-the-art. Since there exist many different ASP solvers we selected some which seem most important to us. In terms of standard solvers we therefore had a look at the ASP competition 2009 and took the best two stand alone ASP solvers from there, namely clasp [8] and DLV [18]. Note that there are several versions and parameter settings for clasp. For our evaluation we are using the package clingo, which is a combination of the grounder gringo [10] and the clasp solving engine. But of course we also want to compare OMiGA to solvers using grounding on-the-fly. To our knowledge there are only two such solvers, namely GASP [27] and ASPERIX [16]. GASP is implemented in SICStus Prolog, a commercial licensed software. Furthermore from [17] one can conclude that it is much slower than ASPERIX. Therefore we only compare our solver to ASPERIX. We furthermore evaluate our solver with and without rewriting, to see how much the rewriting influences OMiGAs calculation.

For our benchmark instances we have chosen problems that represent several different aspects of ASP or have been of relevance for other solvers. The most known problems we use are graph reachability and 3-colourability (3COL). The reachability problem was chosen in order to test the solvers propagation speed, since it is a positive program that can be solved by propagation only. 3COL is a well known NP-complete problem that requires a lot of guessing.

Another two benchmarks where chosen from [17], namely birds and StratProg. Birds is a simple program that consists of several facts (the birds), and a rule with negation that determines if a certain bird flies or not. StratProg is a problem consisting of two guesses, where the second depends on the first. If the first guess is done, the remaining problem becomes stratified and can be solved by propagation only.

The last problem we want to consider is the cutedge program. Here a graph $G = (V, E)$ is given, and the aim is to calculate reachability, after one arbitrary edge has been removed from the graph. The interesting thing about the cutedge program is that there is a single guess at the beginning and the rest of the program can be solved with propagation only. The combination of guessing and propagation often occurs in ASP programs and can be problematic when trying to find an optimal grounding in advance. The cutedge program now shows how grounding on-the-fly can influence such calculations.

For our evaluation a PC with an AMD FX(tm)-8120 Eight-Core Processor 3.10 GHz and 16GB RAM is used. The operating system is a Windows 7 Home Premium with Service Pack 1. For DLV we used the build of 17.12.2012, for clasp we used clingo version 3.0.4 with clasp version 1.3.10 and for ASPERIX we used the latest version 0.2.4. In the following the encoding for each benchmark is presented, an evaluation of several problem instances for all these solvers is given, and we describe our concrete findings.

	reach1	reach2	reach3	reach4
DLV	1,3	0,8	5,8	11,9
Clingo	0,4	0,3	3	6,5
ASPERIX	13	15,7	>600	>600
OMiGA-rw	1	1	6,9	19,51
OMiGA-nrw	0,9	0,9	5,6	10,7

Table 6.1: Graph reachability benchmark evaluation: Showing the runtime in seconds needed to find the single answer set for each solver and benchmark instance

6.1 Graph reachability

In the graph reachability problem a graph $G = (V, E)$ is given, where the graph is represented by edge atoms $edge(x, y)$ where $(x, y) \in E$. The aim of our encoding (6.1) is to find all nodes that are connected to the vertex 10189, which is part of all our benchmark instances. Since reachability is a positive program it can be solved without guessing. Therefore we use the graph reachability benchmark to evaluate our solver's propagation speed. Note that for standard solvers using intelligent grounding the grounding step is sufficient to solve positive programs. Hence in the results we compare OMiGAs propagation speed to the propagation speed of ASPERIX, as well as to the standard solvers grounding speed.

$$reachable(X, Y) :- edge(X, Y). \tag{6.1}$$
$$reachable(X, 10189) :- reachable(X, Z), reachable(Z, 10189). \tag{6.2}$$

Our concrete benchmark instances are taken from the ASP competition 2009. We name the instances reach1-4, where the graph of

- reach1 contains 11711 nodes and 23980 edges,
- reach2 contains 386 nodes and 30342 edges
- reach3 contains 1016 nodes and 341900 edges,
- reach4 contains 2569 nodes and 701598 edges.

The results of Figure 6.1 show that for the graph reachability program OMiGA is 10 times faster than ASPERIX in terms of propagation speed for small instances. For big instances ASPERIX does not find an answer set in reasonable time. This is due to its naive matching approach when finding applicable rules. OMiGAs Rete network on the other side can handle the big amount of propagation occurring in this problem very well. One can see that since OMiGA (without rewriting) is even slightly faster than DLV. Clasp on the other side is still twice as fast as OMiGA. We therefore think that clasp's grounder gringo is really effective concerning positive programs. We furthermore conclude from these results that the currently used rewriting of OMiGA is not optimal, since for reach4 OMiGA with rewriting needs almost twice the time it needs without the rewriting. This is due to our current naive rewriting approach, which generates many unnecessary rules and, for this benchmark, no rules that provide any benefit for solving it.

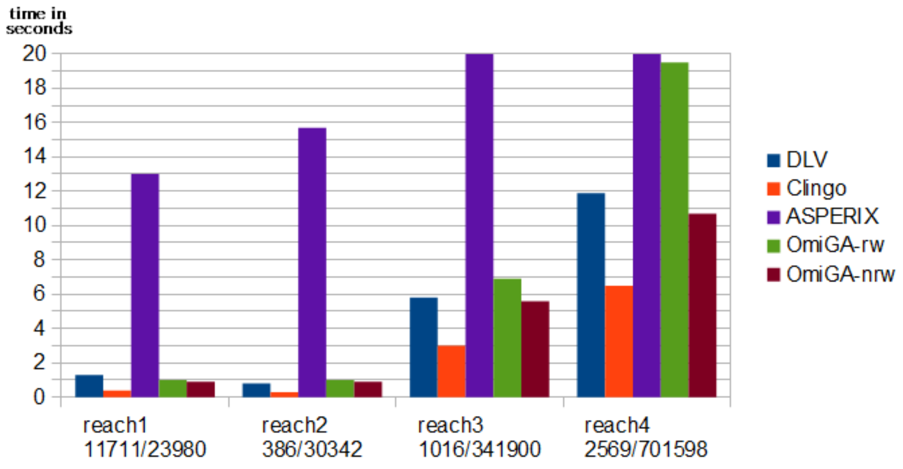

Figure 6.1: Graphical representation of Figure 6.1 for the graph reachability benchmark. x-axis: instances (number of nodes/edges), number of edge atoms increases from left to right, y-axis: runtime in seconds

6.2 Birds

The bird encoding (6.3) consists of several facts $b(X)$, which represent the birds. Some of those birds are either ostriches $o(X)$, penguins $p(x)$ or super-penguins $sp(X)$. Each of those birds flies $f(X)$ or does not fly $nf(X)$. This is determined by a rule that assigns $f(X)$ to any bird $b(X)$ that is not a penguin and not an ostrich. Furthermore every super penguin flies. Another rule assigns $nf(X)$ to a penguins that are not super penguins and to all ostriches. In [17] this problem was used to show that ASPERIX is twice as fast as standard solvers. This was three years ago. Intelligent grounding and standard ASP solvers have evolved in the meantime. Therefore our results can not reproduce this. We use this program to evaluate OMiGAs grounding speed on rules with negative body in combination with closing.

$$p(X) : -sp(X). \tag{6.3}$$
$$b(X) : -p(X). \tag{6.4}$$
$$b(X) : -o(X). \tag{6.5}$$
$$f(X) : -b(X), not\ p(X), not\ o(X). \tag{6.6}$$
$$f(X) : -sp(X). \tag{6.7}$$
$$nf(X) : -p(X), not\ sp(X). \tag{6.8}$$
$$nf(X) : -o(X). \tag{6.9}$$

The size of our concrete benchmark instances can be determined by the number of *bird* facts. For all instances 10% of the birds will be ostriches, another 20% will be penguins where half of them are superpenguins. In the following we consider instances for 10^3, 10^4, 10^5, $2{*}10^5$ and $5{*}10^5$ birds.

	1000	10000	100000	200000	500000
DLV	0,1	0,5	2,6	5	13,3
Clingo	0	0,1	0,9	1,8	4,7
ASPERIX	0	0,4	2,8	5,6	13,8
OMiGA-rw	0,5	1,6	4,6	7,3	19,5
OMiGA-nrw	0,3	1,1	3,7	6,1	11,9

Table 6.2: Birds benchmark evaluation table. Showing for the used benchmark instances the runtime in seconds needed to find the single answer set for each tested solver.

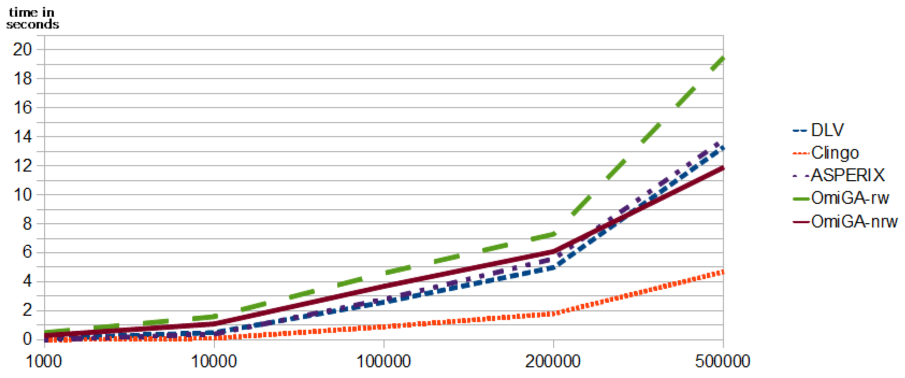

Figure 6.2: Graphical representation of Figure 6.2 for the birds benchmark. x-axis: number of birds increasing from left to right. y-axis: runtime in seconds.

The results of the bird program shown in Figure 6.2 can not confirm the results in [17]. Clingo is about three times faster than all the other solvers, while ASPERIX is slightly slower than DLV and OMiGA (without rewriting). We assume that there have been some major improvements in the grounder components of DLV and clasp since 2009 while ASPERIX was not developed any further than 2010. OMiGA is only slightly slower than DLV for the small instances, while it is slightly faster for the biggest one. This is because our Rete network of course needs some time to initialize, which, in case of the bird example, seems to only pay off for the really big instances. In comparison to ASPERIX, OMiGA is also only slightly faster. From the reachability results one might have expected a bigger gap between those two solvers. This is not the case because the rules of the birds encoding only consist of predicates of arity one. Therefore the matching of rules becomes easy and our Rete network has no advantage over the ASPERIX approach in this benchmark. The reason for clingo being so fast is again its intelligent grounder which does all the work and seems to be very well optimized for these kind of problems.

	10	100	200	300	400	500	1000	2000
DLV	0,2	1,6	11,5	155,4	96,3	200	>900	>900
Clingo	0	0,2	0,5	1,2	2,3	3,6	16,3	OUT_OF_MEMORY
ASPERIX	0,1	0,7	1,9	3,4	6	9,4	55,5	219
OMiGA-rw	0,8	1,1	2,1	3,8	6,2	9,4	37,3	158,6
OMiGA-nrw	0,4	1	2	3,5	6	8,7	34,4	147,8

Table 6.3: StratProg benchmark evaluation table. Showing for the used benchmark instances the runtime in seconds needed to find 100 answer sets for each tested solver.

6.3 StratProg

This program contains one guess on the rules 6.10 and 6.11 which either puts a or b into the answer set. Rules 6.12 and 6.13 have the form of a guess as well and add $aa(X, Y)$ (resp $bb(X, Y)$) to the answer set, for each $a(X)$ (resp $b(X)$) in combination with each element X for which there is no $b(X)$ (resp. $a(X)$). But after the first guess is resolved (so after all a and b facts are derived) the program becomes stratified and rules 6.12 and 6.13 are solvable by propagation only. While solving StratProg is easily done, the stratified part poses a problem for standard solvers when creating the grounding. Since they do not know which rules are guessed in advance, they have to ground much more than needed during the solving phase.

$$a(X) : -p(X), not\, b(X). \tag{6.10}$$

$$b(X) : -p(X), not\, a(X). \tag{6.11}$$

$$aa(X, Y) : -a(X), p(Y), not\, a(Y). \tag{6.12}$$

$$bb(X, Y) : -b(X), p(Y), not\, b(Y). \tag{6.13}$$

The size of our concrete benchmark instances can be measured by the amount of p predicates which determine the number of ground rule instances in this encoding. In the following we will consider instances with 10, 100, 200, 300, 400, 500, 1000 and 2000 p facts. We will first consider a calculation over 100 answer sets and than have another look at a calculation for a single answer set.

The evaluation of the StratProg benchmarks presented in Figure 6.4 shows that for big instances OMiGA is about 30% faster than ASPERIX, while for small instances ASPERIX is up to 0.3 seconds faster. This is due to our Rete network, which does not pay off for small instances, since it needs some time to initialize. One can see that DLV is slower than all the other solvers. This is because DLV can not optimize its grounding for this problem and is trapped in exponential blowup. Note that clasp has a problem with the grounding as well, since most of it's calculation time is needed for building the grounding. Anyway clasp seems to have implemented some heuristic that reduces the size of the StratProg grounding a lot, in comparison to DLV. OMiGA in comparison to DLV is about 20 times faster for the instance with 500 p predicates. For the last two benchmark instances DLV does not yield results in reasonable time. While clingo runs out of memory for the biggest instance, it has the fastest calculation speed of

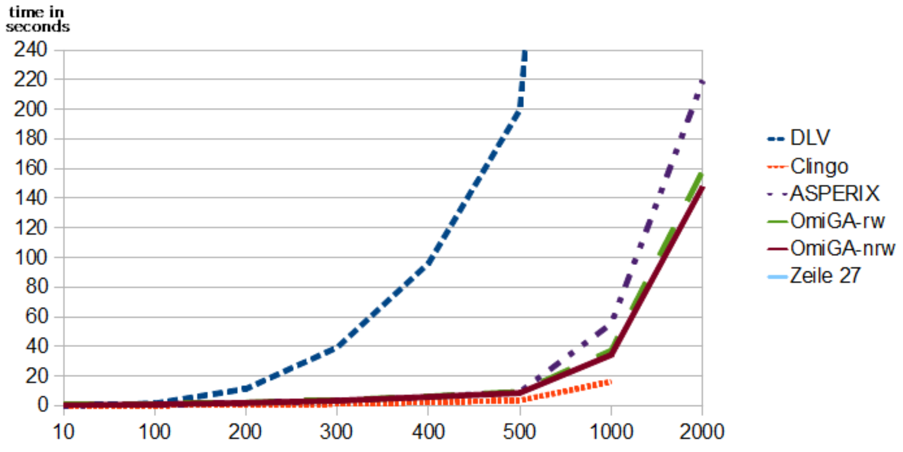

Figure 6.3: Graphical representation of Figure 6.4 for the StratProg benchmark calculation over 100 answer sets. x-axis: instance size increasing by the number of p atoms containd from left to right, y-axis: runtime in seconds

	10	100	200	300	400	500	1000	2000
DLV	0	1,2	10,2	37	90,8	193,5	>900	>900
Clingo	0	0,2	0,5	1,2	2,3	3,6	16,3	OUT_OF_MEMORY
ASPERIX	0	0	0,1	0,1	0,2	0,2	0,5	3,1
OMiGA-rw	0,1	0,3	0,4	0,6	0,8	1	5	20,2
OMiGA-nrw	0,1	0,3	0,4	0,5	0,8	1	4,9	20

Table 6.4: StratProg benchmark evaluation table. Showing for the used benchmark instances the runtime in seconds needed to find one answer set for each tested solver.

all the solvers. The arising OUT_OF_MEMORY-Exception for the last instance can maybe be avoided by using some other configuration of clingo. On the first impression our results seem to differ from [17] but we want to note that we actually can confirm the results of the ASPERIX paper (except for the now more performant speed of DLV and clasp).

When running calculation over only one answer set for our benchmark instances OMiGA is about three times faster than clingo. This is due to the creation of the grounding which seems to take a lot of time of clingo's calculation speed. While this indeed shows a benefit of grounding on-the-fly (we obtain an answer set faster than clingo), clingo is still faster at solving once the grounding has been done. ASPERIX on the other side solves the calculation for one answer set almost instantaneously. This is because the rules of StratProg are easy to match for their naive approach since the contained atoms are only of arity one.

6.4 Cutedge

The Cutedge encoding 6.14 is a variation of the reachability program. Like in the graph reachability problem a Graph G is given which represented by edge atoms. But rather than directly calculating reachability, a guess is made first (line 1) which deletes one edge from the graph. Then reachability (line 4 and 5) is calculated over the remaining edges (line 2 and 3). The interesting thing about the Cutedge problem is that it consists of a single guess followed by a heavy propagation step. Since for general ASP programs most guesses are followed by propagation, we want to show with the cutedge program that this can pose a huge challenge for a standard solver and we want to show that grounding on-the-fly can indeed be a useful approach to that problem. Finally we want to note that OMiGA with rewriting is not much worse than OMiGA without rewriting. We conclude that our rewriting indeed adds some useful rules that reduce the number of overall guesses, but still creates too much overhead to be beneficial.

$$delete(X,Y) : -edge(X,Y), not\, keep(X,Y). \tag{6.14}$$
$$keep(X,Y) : -edge(X,Y), delete(X1,Y1), X1! = X. \tag{6.15}$$
$$keep(X,Y) : -edge(X,Y), delete(X1,Y1), Y1! = Y. \tag{6.16}$$
$$reachable(X,Y) : -keep(X,Y). \tag{6.17}$$
$$reachable(X,98) : -reachable(X,Z), reachable(Z,98). \tag{6.18}$$

Our concrete benchmark instances are named cut1-8, where the graph of

- cut1 contains 100 nodes and 2887 edges
- cut2 contains 100 nodes, 4863 edges
- cut3 contains 100 nodes, 5903 edges
- cut4 contains 1000 nodes, 198122 edges.
- cut5 contains 11711 nodes and 23980 edges,
- cut6 contains 386 nodes and 30342 edges
- cut7 contains 1016 nodes and 341900 edges,
- cut8 contains 2569 nodes and 701598 edges,

Note that the graphs of cut5-8 correspond one-to-one to those of reach1-4 of the graph reachability problem (6.1).

The evaluation of the cutedge program is given in Figure 6.5; it is interesting for several reasons. First we can see that only ASPERIX and OMiGA are capable of solving all the cutedge benchmark instances. This is due to the grounding on-the-fly approach both solvers use. (Note that while ASPERIX was not able to find 10 answer sets in reasonable time, it did find at least one answer set within 32400 seconds.) The other solvers need too much time and space to create

	100/2887	100/4863	100/5903	11711/23980	386/30342
DLV	371,6	1082,6	1635,7	OUT_OF_MEMORY	OUT_OF_MEMORY
Clingo	31	92,3	136,95	OUT_OF_MEMORY	OUT_OF_MEMORY
ASPERIX	23,9	56,1	80	>3000	>3000
OMiGA-rw	3,6	8,2	8,4	90	157,3
OMiGA-nrw	2,3	4,3	5,6	84,1	143,7

Table 6.5: Cutedge benchmark evaluation table. Showing for the used benchmark instances the runtime in seconds needed to find 10 answer sets for each tested solver.

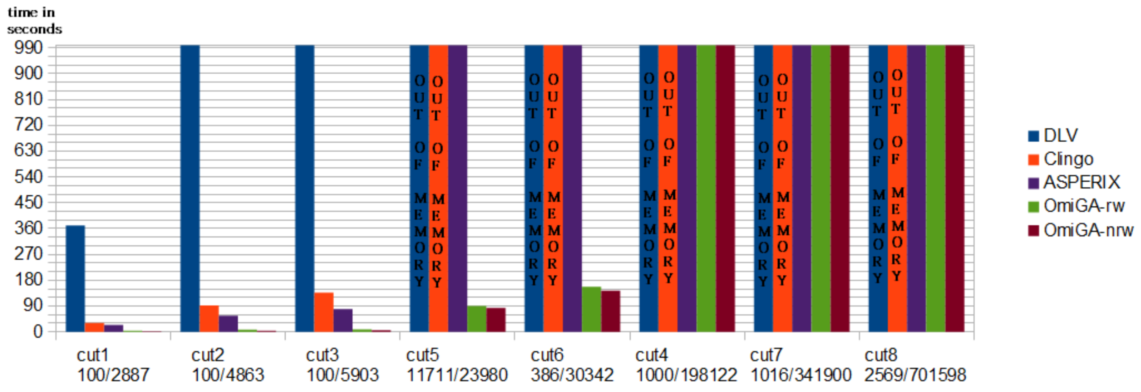

Figure 6.4: Graphical representation of Table 6.5 and Table 6.6 for the Cutedge benchmark calculation over 10 answer sets. x-axis: instances, number of edge atoms increases from left to right, y-axis: runtime in seconds

	1016/341900	2569/701598	1000/198122
DLV	OUT_OF_MEMORY	OUT_OF_MEMORY	OUT_OF_MEMORY
Clingo	OUT_OF_MEMORY	OUT_OF_MEMORY	OUT_OF_MEMORY
ASPERIX	>32400	>32400	>32400
OMiGA-rw	2917	10264	9298
OMiGA-nrw	2633	9681	8686

Table 6.6: Cutedge benchmark evaluation (Figure 6.5 extended by huge instances). Showing for the used benchmark instances (number of nodes/edges) the runtime in seconds needed to find 10 answer sets for each tested solver.

their grounding and never reach their solving phase because they run out of memory. Therefore this benchmark shows clearly the benefits of a grounding on-the-fly approach. Furthermore one can also see that even for the three instances that DLV and clasp can solve, OMiGA is about 300 times faster than DLV and about 16 times faster than clingo. Like in the StratProg evaluation clingo runs out of memory for the big instances.

6.5 3COL

3COL is a well known NP complete problem, where a graph G = (V,E) is given. The goal is to assign one of three colors to each vertex such that there is no edge that connects two vertices of same color. The standard encoding depends on guessing (line 6.19 to 6.21) and three constraints (line 6.22 to 6.24). One might notice that there are rules that can be simply propagated, therefore finding an answer set depends mostly on the time a solver needs for guessing and backtracking.

$$blue(N) :- node(N), not\ red(N), not\ green(N). \tag{6.19}$$
$$red(N) :- node(N), not\ blue(N), not\ green(N). \tag{6.20}$$
$$green(N) :- node(N), not\ red(N), not\ blue(N). \tag{6.21}$$
$$:- edge(N1, N2), blue(N1), blue(N2). \tag{6.22}$$
$$:- edge(N1, N2), red(N1), red(N2). \tag{6.23}$$
$$:- edge(N1, N2), green(N1), green(N2). \tag{6.24}$$

The instances used for our evaluation are three-color able and of variable density. They were generated randomly, since common benchmark instances are too big for OMiGA an ASPERIX to solve. Their size is given in terms of the graph to be colored:

- 10 nodes, 36 edges
- 20 nodes, 32 edges
- 20 nodes, 57 edges
- 20 nodes, 76 edges
- 20 nodes, 78 edges
- 20 nodes, 102 edges
- 20 nodes, 143 edges
- 30 nodes, 123 edges

When looking at our benchmark instances for the 3Col problem one might notice that they are very small. In Figure 6.7 which shows our evaluation one can therefore see that the standard solvers, DLV and Clingo can solve those instances in almost no time. Anyway these instances

	10/36	20/32	20/57	20/78	20/102	20/76	20/143	30/126
DLV	0	0	0	0	0	0	0	0
Clingo	0	0	0	0	0	0	0	0
ASPERIX	2	0,2	15000	7342	2197	7973,7	325,4	>9000
OMiGA-rw	0,5	0,5	3,7	1	1,2	1,5	0,9	50,3
OMiGA-nrw	0,1	0,1	2,3	0,8	0,7	1,1	0,5	31

Table 6.7: 3COL benchmark evaluation table. Showing for the used benchmark instances the runtime in seconds needed to find up to 100 answer sets for each tested solver.

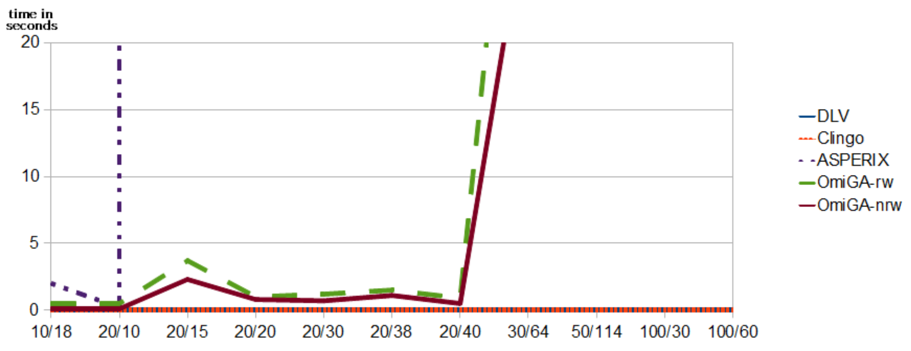

Figure 6.5: Graphical representation of Table 6.7 for the 3COL benchmark calculation over up to 100 answer sets. x-axis: instances, graph size increases from left to right, y-axis: runtime in seconds

already pose a challenge to us since OMiGA neither has heuristics for guessing nor implemented learning algorithms. Anyway OMiGA is still a thousand times faster than ASPERIX for the 3COL benchmarks. The order in which guesses are done by ASPERIX and OMiGA should be about the same, therefore this huge difference in runtime again seems to arise from our Rete network which can track down applicable rule instances faster than the ASPERIX approach. Note that unsatisfiable instances show no different runtime behavior, since some of the given instances lead to less than 100 answer sets, and therefore all possible answer sets have to be checked anyway (like in the unsatisfiable case). The calculation over only one answer set is not conclusive since some cases yield much better runtime than others since the algorithm checked that one first. In any case even if the algorithm was lucky, it's still much worse than clasp and DLV, simply because of initializing the Rete network needs time.

6.6 Summary

Our evaluation shows that grounding on-the-fly can indeed influence calculation time to the better, as one can see in the StratProg and especially in the Cutedge benchmark. While clasp is faster than ASPERIX and OMiGA when calculating 100 answer sets of the StratProg benchmark, ASPERIX and OMiGA find their first answer set before clasp even starts its solving phase, because it is still creating its grounding. For the cutedge benchmark, one can see that the standard solvers are not capable of solving the bigger instances. This is due to the huge grounding they need to create. Both DLV and clasp run out of space far before finishing grounding and even for those instances where they finish their grounding phase overall they are more than 10 times slower than OMiGA. This shows that grounding on-the-fly indeed is a good approach to address the problem of exponential blowup.

Furthermore we have shown that OMiGA has good propagation speed, since we are faster than ASPERIX and DLV in the graph reachability program. While clasp indeed is faster than OMiGA in the graph reachability benchmark one has to note that this is the intelligent grounding component of clasp, not the actual speed of clasp for propagating rules while solving. We think it is already a good result that our propagation speed can keep up with DLVs grounding speed, but we do think that it can still be increased by optimizing our Rete network.

Furthermore our evaluation for the 3COL problem has shown that OMiGA needs optimizations concerning its choice unit, including heuristics and learning strategies, since it can neither keep up with clasp nor DLV. Anyway, OMiGA is still much faster than ASPERIX for the 3COL problem. Overall we think that OMiGA shows that grounding on-the-fly indeed is viable approach for answer set programming.

CHAPTER 7

Related Work

In this section we discuss some of the main differences and similarities between OMiGA and current state-of-the-art ASP solvers, as well as OMiGAs relations to some Rete technology.

ASPERIX [16, 17] is a solver using the grounding on-the-fly principle like OMiGA. Since we relied heavily on the work that was done around ASPERIX, we also have a whole section for ASPERIX, see 2.3. Rules are dynamically grounded on demand and applicable rules are applied until an answer set is reached. ASPERIX is written in C++ and uses a rather naive matching algorithm to build ground rule instances, which leads to much recalculation. The main difference between OMiGA and ASPERIX is that OMiGA uses a Rete network to do this, therefore avoiding this kind of recalculation. GASP [27] is another grounding on-the-fly solver and uses the same theoretical background as ASPERIX and OMiGA. GASP is implemented in SICStus Prolog 4 [1] and uses Constraint Logic Programming over finite domains. At each local grounding step, a constraint satisfaction problem (CSP) is built from the rules body atoms and then solved to determine which rules can be applied. GASP supports cardinality constraints, but different from OMiGA it does not accept programs with function terms.

There are many standard ASP solvers out there, among them DLV [18], CMODELS [19], SMODELS [23, 25, 32], clasp [8, 9], and ASSAT [20]. All of them use a two step approach where the input program is first grounded and then in the second step the resulting propositional program is solved. Note that the grounding is done by a separate unit that uses intelligent grounding [6, 10, 21], an important technique for standard solvers. The solving can be done in many different ways. Clasp uses a conflict driven approach by turning parts of the propositional program first into a boolean constraints (so called nogoods). Then they use further preprocessing by using some techniques they adapted from SAT solving. The solving is then done by combining propagation and nogoods. In case of a derived contradiction additional nogoods are learned. Furthermore clasp uses sophisticated heuristics to determine where to guess as well as restarts

[1] http://www.sics.se/isl/sicstuswww/site

in order to check the most promising parts of the input program first. ASSAT and CMODELS completely transform the input program into a SAT program. By using loop-formulas [14,34,35] they are able to do this without adding variables but still having a one-to-one correspondence between the answer sets of the ASP input program and the solutions to the SAT problem. Note that the number of loop-formulas can be exponential and they are created iteratively when needed. DLV on the other side directly works with the propositional rules and applies them until an answer set is reached. For this purpose DLV uses sophisticated heuristics for choosing rules to guess on and for back jumping. The SMODELS procedure for answer set computation builds upon so called primitive rules. First the input program is rewritten into a ground program without variables. From this ground program all rules are translated into primitive rules [24]. On those primitive rules a Davis-Putnam [31] like procedure finally computes the stable models.

Aside from finding answer sets for a given ASP program those solvers have several features and support different functions that make coding an ASP program a lot easier. All these solvers support aggregate functions, disjunction and a restricted form of function terms. Weak and weight constraints are supported by DLV, CMODELS, SMODELS and clasp. DLV furthermore offers a planning, diagnosis and inheritance front-end. OMiGA on the other side only supports function terms at the moment, but in an unrestricted form. DLV, CMODELS, SMODELS, CLASP as well as ASSAT are implemented in C++. Since we furthermore do not know of any other solvers that are implemented in Java, we think that OMiGA is currently the only ASP solver implemented completely in Java. Note that DLV offers a Java wrapper [3] but is not implemented in Java itself.

The idea behind OMiGAs Rete network comes from rule based production systems [4]. A rule-based production system consists of production rules which are conceptually similar to those of an ASP program. They have a different syntax that resembles to that of an if statement from programming languages:

$$IF < condition > THEN < action >$$

Where $condition$ most of the time is a pattern that can be matched against a database and action can be any arbitrary procedure that most of the time adds new atoms to the database or modifies old atoms. While this offers a lot more functionality than plain ASP does, the idea behind is the same: if some condition holds then something happens. So for our purpose one can see the condition as an ASP rules body and the action as its head.

Anyway there is a huge difference in the way non-monotonic rules are treated. Rather than guessing on rules, the rules within such a system are prioritized, and in case of more rules being applicable at the same time the one with highest priority is fired. Because of this, the calculation of a rule based production system is deterministic and, different from ASP programs, there is always only one solution (rather than multiple answer sets). Furthermore they have another way to deal with inconsistency, since rules are also allowed to remove already derived facts from the knowledge base. Therefore, when a fact is added that would lead to inconsistency, the old fact is simply removed from the database or some function is called that will solve this conflict. Nevertheless, as OMiGA has shown that their underlying data structure - the Rete network - can also be adapted to help calculating ASP programs. There are many relevant rule based engines out there like for example CLIPS [13] and Soar [15]. The most popular rule based engine for Java is DROOLS [30] as part of the JBOSS project [2]. Note that we did not use an already existent rule based system like DROOLS, because our first test yielded that, while they indeed do offer many high level functions, they create significant overhead for our purpose. Therefore we implemented our own Rete network which is specialized to work with ASP rules by propagating variable assignments. To further improve OMiGA's Rete network the following papers [26, 28, 29, 33] offer several ideas of how to optimize the network's join order as well as how to traverse the network efficiently.

[2]http://www.jboss.org/drools

CHAPTER 8

Conclusion

In this thesis we introduced the new grounding on-the-fly ASP solver called OMiGA, which uses a Rete network to store partial rule groundings. This gives OMiGA a big advantage over other grounding on-the-fly solvers which use another approach, and therefore need to recalculate partial rule groundings whenever a new fact is derived. We presented the theoretical algorithm behind OMiGA in Chapter 3 and proof of soundness and completeness. Within Chapter 5 we gave a detailed description of our implementation. Furthermore we also conducted an experimental evaluation to compare OMiGA to other ASP solvers on several benchmarks in Chapter 6.

OMiGA is a state-of-the-art grounding on-the-fly ASP solver. While standard ASP solvers like DLV [18] or Clasp [8] use a two step approach where they first ground the whole input program and then solve the grounded one, OMiGA is built upon the concept of grounding on-the-fly like ASPERIX [16] and GASP [27]. Rather than first grounding the whole input program, a solver using grounding on-the-fly creates the needed grounding on demand while solving, therefore interleaving grounding and solving. Since a grounding on-the-fly solver can ground with respect to the current partial interpretation, it can create smaller groundings than a standard solver. Nevertheless, finding applicable ground rule instances for nonground rules is challenging. To address this issue OMiGA utilizes a Rete network [7], like it is normally used within rule based production systems [30]. OMiGA utilizes its Rete network to ground rules step by step and save all calculated partial rule groundings. This speeds up the propagation phase of our solver as well as enable us to easily determine all available choices left. A detailed description of our Rete network is given in Section 5.2.

Evaluation: Our evaluation shows that OMiGA has indeed a very good propagation speed, since it is more than 10 times faster than ASPERIX in the graph reachability benchmark and can even compete with the calculation speed of the standard solver DLV. We think that handling nonground ASP rules during solving is not a disadvantage and therefore we think that grounding on-the-fly solvers can someday overcome standard solvers. Additionally there are problem instances a standard solver can not handle because of the exponential blow up occurring during the pregrounding. The Cutedge benchmark is one such example. Here DLV and clasp do not find a solution for the huge instances and even for the small ones OMiGA is more than 15 times faster (finishing calculation before DLV and clasp finished their grounding step). While OMiGA is about 10 times faster than ASPERIX for small cutedge instances, ASPERIX seems to have trouble with propagation and its calculation speed gets worse the bigger the problem instances get. But of course in terms of general non-monotonic programs OMiGA can not keep up with standard solvers, as one can see in the results for the 3COL evaluation. Nevertheless OMiGA is three orders of magnitude faster than ASPERIX in the 3COL benchmarks.

Note that the main focus of the practical part of this thesis is the combination of grounding on-the-fly with a Rete network. While this indeed seems to be a way to deal with nonground rules, OMiGA lacks several other important things standard ASP solvers have, like heuristics and learning strategies. Both can have a huge impact on the number of choices that have to be done to find an answer set. The only optimization OMiGA uses concerning a reduction of the number of choices done, is the closing of strongly connected components (SCC), an optimization all the other solvers use as well. Since the number of choices exponentially influences the number of possible solutions the solver has to check, OMiGA is not efficient at solving instances where many choices have to be considered. Anyway our evaluation shows that OMiGA indeed can handle nonground rules in a performant way. Optimizing OMiGA is left for future work.

Future work: A huge challenge is to optimize OMiGA, but we are confident that grounding on-the-fly solvers can outperform standard solvers, when adequate ways for learning and heuristics have been developed. In the following we discuss parts of OMiGA where we see potential optimizations.

The most basic improvement can probably be achieved within the rewriting of the input program. At the moment the rewriting is mainly used to rewrite the input program in such a manner that all non-monotonic rules have unique heads. To achieve this, all rules are rewritten at the moment, even those that already have a unique head. This leads to an unnecessary overhead, which impacts the propagation in the Rete network. But this can be fixed easily.

At the moment the Rete network is built in such a way that when considering a rule the first left literal is joined with the second left one and so on until the whole rule body is joined. One might optimize this by building joins in a different order and evaluating atoms first that minimize the workload for the network. Furthermore our current Rete network stores all variable assignments of the partial assignment for the corresponding node, but actually variable assignments that are not needed to unify the head of the rule or for further joins can be dropped in order to save space. So for example for a rule $head(X,Y)$:- $q(X,Z), p(Z,Y), s(X,Y)$., the join node $qp(X,Z,Y)$ does not need to store the value for Z, since the head only needs X and Y for instantiation. Furthermore one might add parallelization to the Rete network. There has already been much research done for the Rete algorithm [7,28,29,33]. What is left to do is adapting this knowledge for ASP and integrating it into our Rete network

Finally the biggest performance gain can probably be achieved within the choice node. Since the runtime is influenced exponentially by the number of choices, each avoided choice is a huge benefit. At the moment a choice is done like this: the first ground rule instance within the first non empty choice node of the current SCC is picked and a guess is performed on it. There is no clever criterion for choosing such rules, but of course the order in which choices are done has a huge impact on the overall number of choices needed. More research is necessary to determine a heuristic that leads to good results.

Another way to optimize the choice unit, and probably the most promising optimization for OMiGA, is learning. One can distinguish between two different kinds of learning. On the one hand we can learn ground rule instances to avoid single conflicts. On the other hand one might learn nonground rules from analyzing the structure of nonground rules that lead to conflicts. We think that learning can increase OMiGAs runtime tremendously but much more research has do be done in this area.

Bibliography

[1] Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, NY, USA, 2003.

[2] Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors. *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*. Springer, 2007.

[3] Francesco Buccafurri, editor. *2003 Joint Conference on Declarative Programming, AGP-2003, Reggio Calabria, Italy, September 3-5, 2003*, 2003.

[4] Bruce G. Buchanan and Richard O. Duda. Principles of rule-based expert systems. *Advances in Computers*, 22:163–216, 1983.

[5] Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors. *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*. Springer, 2009.

[6] Wolfgang Faber, Nicola Leone, and Simona Perri. The intelligent grounder of DLV. In Esra Erdem, Joohyung Lee, Yuliya Lierler, and David Pearce, editors, *Correct Reasoning*, volume 7265 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2012.

[7] Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[8] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. *clasp* : A conflict-driven answer set solver. In Baral et al. [2], pages 260–265.

[9] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. The conflict-driven answer set solver clasp: Progress report. In Erdem et al. [5], pages 509–514.

[10] Martin Gebser, Torsten Schaub, and Sven Thiele. GrinGo : A new grounder for answer set programming. In Baral et al. [2], pages 266–271.

[11] Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors. *Logic Programming and Nonmonotonic Reasoning, 5th International Conference, LPNMR'99, El Paso, Texas, USA, December 2-4, 1999, Proceedings*, volume 1730 of *Lecture Notes in Computer Science*. Springer, 1999.

[12] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.

[13] Joseph C. Giarratano and Gary D. Riley. *Expert Systems: Principles and Programming*. Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 2005.

[14] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, 2006.

[15] John E. Laird, Allen Newell, and Paul S. Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

[16] Claire Lefèvre and Pascal Nicolas. A first order forward chaining approach for answer set computing. In Erdem et al. [5], pages 196–208.

[17] Claire Lefèvre and Pascal Nicolas. The first version of a new asp solver : ASPeRiX. In Erdem et al. [5], pages 522–527.

[18] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[19] Yuliya Lierler. cmodels - SAT-based disjunctive answer set solver. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *LPNMR*, volume 3662 of *Lecture Notes in Computer Science*, pages 447–451. Springer, 2005.

[20] Fangzhen Lin and Yuting Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[21] Guohua Liu and Jia-Huai You. Lparse programs revisited: Semantics and representation of aggregates. In Maria Garcia de la Banda and Enrico Pontelli, editors, *ICLP*, volume 5366 of *Lecture Notes in Computer Science*, pages 347–361. Springer, 2008.

[22] Victor W. Marek, Ilkka Niemelä, and Miroslaw Truszczynski. Origins of answer set programming - some background and two personal accounts. *CoRR*, abs/1108.3281, 2011.

[23] Ilkka Niemelä and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *LPNMR*, volume 1265 of *Lecture Notes in Computer Science*, pages 421–430. Springer, 1997.

[24] Ilkka Niemelä, Patrik Simons, and Timo Soininen. Stable model semantics of weight constraint rules. In Gelfond et al. [11], pages 317–331.

[25] Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen. Smodels: A system for answer set programming. *CoRR*, cs.AI/0003033, 2000.

[26] Tugba Özacar, Övünç Öztürk, and Murat Osman Ünalir. Optimizing a rete-based inference engine using a hybrid heuristic and pyramid based indexes on ontological data. *JCP*, 2(4):41–48, 2007.

[27] Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. GASP: Answer set programming with lazy grounding. *Fundam. Inform.*, 96(3):297–322, 2009.

[28] Mark Perlin. Scaffolding the rete network. *Tools for Artificial Intelligence, 1990.,Proceedings of the 2nd International IEEE Conference on*, pages 378–385, 1990.

[29] Mark Perlin. Topologically traversing the rete network. *Applied Artificial Intelligence*, 4(3):155–177, 1990.

[30] Mark Proctor. Drools: A rule engine for complex event processing. In Andy Schürr, Dániel Varró, and Gergely Varró, editors, *AGTIVE*, volume 7233 of *Lecture Notes in Computer Science*, page 2. Springer, 2011.

[31] Patrik Simons. Extending the stable model semantics with more expressive rules. In Gelfond et al. [11], pages 305–316.

[32] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[33] Murat Osman Ünalir, Tugba Özacar, and Övünç Öztürk. Reordering query and rule patterns for query answering in a rete-based inference engine. In Mike Dean, Yuanbo Guo, Woochun Jun, Roland Kaschek, Shonali Krishnaswamy, Zhengxiang Pan, and Quan Z. Sheng, editors, *WISE Workshops*, volume 3807 of *Lecture Notes in Computer Science*, pages 255–265. Springer, 2005.

[34] Yisong Wang, Jia-Huai You, Li-Yan Yuan, and Yi-Dong Shen. Loop formulas for description logic programs. *TPLP*, 10(4-6):531–545, 2010.

[35] Yisong Wang, Ying Zhang, and Mingyi Zhang. Constructing first-order loops of normal logic programs. In *FSKD*, pages 352–356. IEEE, 2011.