An Interactive Approach to Answer Set Programming

Martin Brain[1], Richard Watson[2], and Marina De Vos[1⋆]

[1] Department of Computer Science, University of Bath Bath, United Kingdom
{mjb,mdv}@cs.bath.ac.uk
[2] Computer Science Department, Texas Tech University, Lubbock, Texas, US
rwatson@cs.ttu.edu

Abstract. This paper outlines a novel approach to the computation of answer sets in an evolving and interactive environment. Instead of recomputing the semantics of the entire program, our approach updates the answer sets after incremental changes to the rule base. This is intended for application domains in which the answer set program is developed or updated during the run time of the system but answer sets are required throughout. In this paper we focus on the theoretical background and presentation of the algorithm for handling the addition.

1 Introduction

$AnsProlog^*$ or *Answer Set Programming* (ASP)[4] is a modern logic system designed for knowledge representation, reasoning and logic programming. Key aims in its development are expressive power, simplicity of use, ease and simplicity of expression and capacity for efficient implementation. The semantics of an $AnsProlog^*$ program is defined in terms of *answer sets*. Informally, these are sets of atoms that are consistent with the rules of a program and supported by a deductive process. A number of tools exist which compute the answer sets of a given program [8, 9]. For many application domains these tools are unsuitable as they have to repeat the entire computation for any and all changes to the input program, no matter how trivial. For example, in a system that uses answer set programs to represent the knowledge of agents, it is unacceptable to have a costly recomputation process after each piece of information is learnt.

In this paper we discuss theoretical mechanisms to track the impact of changes to answer sets when a program is updated. These mechanisms will form the core of our algorithm.

It is important to note that the presented algorithm in no way changes the syntax or the semantics of the programs. The answer sets of a new rule set given by the incremental algorithm is exactly the same as the one given by a traditional answer set solver. The algorithms also do not attempt to produce a solution method of lower theoretical complexity; in the worst case scenario all answer sets have to be recomputed. Due to space restrictions, proofs have been omitted but can be found in [2].

⋆ This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging technologies under the IST-2001-37004 WASP project.

2 The Answer Set Semantics

For compactness and convenience of notation we limit ourselves to an overview of traditional ASP[3]. A more in depth coverage is presented in [4]. What is discussed here is also a syntactic and functional subgroup of $AnsProlog^*$, referred to as $AnsDatalog^{\perp}$.

An $AnsDatalog^{\perp}$ program is made up of a set of *rules*. Each rule has the form: $A_0 \leftarrow A_1, \ldots, A_n, \text{not } A_{n+1}, \ldots, \text{not } A_m.$ where A_0 is an *atom* or $\perp$ and A_i for $i \in [1, m]$ are also atoms. Literals are atoms or negated atoms[3]. A_0 is the *head* of the rule, denoted $H(r)$ for rule r and $\{A_1, \ldots, A_m\}$ is the *body*, denoted $B(r)$. The intuition for this rule is that if all of $A_1, \ldots, A_n$ are known and none of $A_{n+1}, \ldots, A_m$ are known then A_0 is considered to be known (in the case that A_0 is $\perp$, this indicates a contradiction). The set of all atoms appearing in a program P is referred to as the *Herbrand Base*, notated $\mathcal{B}_P$.

When speaking about the status of rules with respect to a given set of atoms the terms *applicable* and *applied* are used. A rule is said to be applicable with respect to a set if all of $A_1, \ldots, A_n$ and none of $A_{n+1}, \ldots, A_m$ are in the set. It is applied if it is applicable and A_0 is also in the set.

Variables are not discussed in this paper for reasons of compactness. In both theory and practical applications of ASP it is assumed that they are removed by an instantiation (or grounding) phase. A formalised treatment of variables in ASP is provided in [4] and [2] discusses our algorithms for handling incremental updates to the instantiation phase.

In this paper we use the characterisation of answer set semantics given by [1]. This is divided into two phases, the semantics of programs that do not contain negation and a semantic criterion and reduct for removing negation.

Programs without negation (also referred to as $AnsDatalog^{-\mathbf{not}}$) each have one answer set which is given by the logical closure of the rule set, i.e. starting with the facts (rules that have no body and are thus not dependent on anything), recursively build a set of anything that can be concluded using a rule who's body is in the set.

To remove negation the Gelfond-Lifschitz reduct (or transformation)[3] is used. To reduce a program with respect to a set of atoms S:

- Removing every rule that contains $\text{not } p$ in the body if $p \in S$
- Removing all remaining negative literals (i.e. $\text{not } q$) from the rules

The answer sets of the program are the sets of atoms S such that S is the answer set of the reduced program. Constraints (rules with $\perp$ in the head) then remove any answer sets in which they are applicable.

Given the program $\{a \leftarrow b.; c \leftarrow \text{not } d, a; d \leftarrow \text{not } c, a; b.; e \leftarrow d.\}$, with answer sets $\{a, b, c\}$ and $\{a, b, d, e\}$, there are some rules that can be added and removed with minimal effort. For example, adding $e \leftarrow b.$ will only add e to every answer set that doesn't already contain it as e is not used in the body of any rule. Likewise removing $e \leftarrow d.$ will have a minimal effect. Adding $\leftarrow e, d.$ to the initial program would simply remove the second answer set. Removing the first rule, $a \leftarrow b$, would cause major changes as the exclusive choice between c and d will not be made.

[3] In this paper we only consider negation as failure, classical negation can easily be simulated.

3 Theorerical Concepts

These are presented separately to increase both the clarity of the following algorithms.

A rule is self referential if it directly refers to itself, i.e. the head of the rule appears in it's body. For example $a \leftarrow a, b.$ or $b \leftarrow \text{not } b, c.$ It is formalised as:

Definition 1. *A rule r is positively self referential if:* $H(r) \in B(r)$.
r is negatively self referential if $\text{not } H(r) \in B(r)$.

Self referential rules pose something of a problem when combined with ideas of the implication and dependency developed later. In the absence of self referential rules deciding whether a rule applies in a given context and how it changes the context can be handled independently, making the computation of answer sets much simpler. Thus they must be removed before the start of the algorithm. Fortunately this can be done with a simple mapping.

Definition 2. *Let P be a program. We define rsr ('Remove Self referential Rules') as*

$$rsr(P) = \{r \in P | r \textit{ non self referential}\} \cup \{\bot \leftarrow B(r). \,|\forall r \in P \cdot\, r \textit{ negatively self referential}\}$$

This mapping is essentially the intuitive treatment of self referential rules; positive self referential rules are removed as they do not add to the information known at any point and negative self referential rules become constraints.

Proposition 1. *Let P be a program and $A \subset \mathcal{B}_P$, then:*

$$A \textit{ is an answer set of } P \;\Leftrightarrow\; A \textit{ is an answer set of } rsr(P)$$

At several points in the presented algorithms there is a requirement to search for answer sets of a program that fulfill certain conditions.

Proposition 2. *Let S be an answer set of the a program P and $A = \{a_1, \ldots, a_n, \text{not } a_{n+1}, \ldots, \text{not } a_m\}$ be a set of literals s.t. $\{a_1, \ldots a_n\} \subset S$ and $\{a_{n+1}, \ldots a_m\} \cap S = \emptyset$. Then S is also an answer set of the augmented program $P * A$:*

$$P * A = P \cup \{a_1 \leftarrow .\,, \ldots,\; a_n \leftarrow .\} \cup \{\bot \leftarrow a_{n+1}.\,, \ldots,\; \bot \leftarrow a_m.\}$$

By computing the answer sets of the augmented program we form a list of all of the answer sets of the original program which satisfy the given criteria.

This technique is of interest as it can be implemented in a fast and efficient manner using existing solver algorithms.

Definition 3. *Let P be a set of rules and $a \in \mathcal{B}_P$ then:*

$$\begin{aligned} support(a, P) &= \{r \in P | H(r) = a\} \\ usage(a, P) &= \{r \in P | a \in B(r) \;\vee\; \text{not } a \in B(r)\} \end{aligned}$$

Support and usage can be used to easily infer some basic facts; for example if $support(a, P) = \emptyset$ then a will not appear in any answer set.

Unfortunately adding rules to a rule set is not as simple as just adding the head atom in answer sets where the rule is applicable. Adding atoms to an answer set may in turn cause or prevent other rules being applicable and thus require other atoms to be added

or removed. These are the *implications* of adding / removing the original atom. The positive implications are the atoms that have to be added and the negative implications are the atoms that need to be removed. For a program P and set of atoms A we define $P_{A,+}(P_{A,-}) \subset P$ be the set of rules that are (not)applicable with respect to A.

Definition 4. *Let P be a program, $A \subset \mathcal{B}_P$ and $a \in \mathcal{B}_P \setminus A$. The positive and negative implications of adding a to A are defined as:*

$$I(+,P,A,+a) = \{H(r) | \exists r \in usage(a,P) \cdot r \in P_{A,-} \wedge r \in P_{A \cup \{a\},+}\}$$
$$I(-,P,A,+a) = \{H(r) | \exists r \in usage(a,P) \cdot r \in P_{A,+} \wedge r \in P_{A \cup \{a\},-}\}$$

Intuitively these are the set of heads of any rule that becomes applicable by adding a and anything that can only be concluded by a rule depending on $\mathrm{not}\ a$.

Definition 5. *Let P be a program, $A \subset \mathcal{B}_P$ and $a \in A$. The positive and negative implications of removing a from A are defined as:*

$$I(+,P,A,-a) = \{H(r) | \exists r \in usage(a,P) \cdot r \in P_{A,-} \wedge r \in P_{A-\{a\},+}\}$$
$$I(-,P,A,-a) = \{H(r) | \exists r \in usage(a,P) \cdot r \in P_{A,+} \wedge r \in P_{A-\{a\},-}\}$$

These are the heads of any rule that become applicable by removing a and any atom that can only be obtained by a rule depending on a.

For compactness the following notation is introduced:

$$I(P,A,+a) := (\ I(+,P,A,+a), I(-,P,A,+a)\ )$$
$$I(P,A,-a) := (\ I(+,P,A,-a), I(-,P,A,-a)\ )$$

Handling implications is a non trivial problem as each one can give rise to further implications. They may also be linked, so order of resolution can matter. For example given the program $P = \{a \leftarrow c, \mathrm{not}\ b; b \leftarrow c, \mathrm{not}\ a.; c \leftarrow d.\}$, which has only one answer set $A = \emptyset$, adding the rule $d \leftarrow$ gives the following sequence of implications: $I(P,\emptyset,+d) = (\{c\},\emptyset)$; $I(P,\{d\},+c) = (\{a,b\},\emptyset)$. Adding d leads to a further implication, c must be added. However adding c leads to a more complex situation. Adding a will stop b being an implication and vica versa. The final answer sets of $P \cup \{d \leftarrow\}$ are $\{a,c,d\}$ and $\{b,c,d\}$. This example shows that implications are the main complexity issue in incremental algorithms.

Dependency graphs showing the relation between rules are a common tool in answer set semantic research. Past uses have included the theoretical basis for stratified logic programs and some other functional subclasses of $AnsProlog^*$ and as the basis for computing answer sets[5]. They have been characterised in a variety of ways[4]. We will use the following definition.

Definition 6. *Let P be a program, it's dependency graph $D = (N,L)$ where N is a set of nodes and $L \subset N \times N \times P$ is a set of directed links annotated with rules, is constructed as follows:*

$$N = \{n_a | \forall a \in \mathcal{B}_P\}$$
$$L = \{(n_a, n_b, r) | \forall r \in P \cdot usage(a,P) \cap support(b,P)\}$$

[4] The link between rules, heads and bodies is essentially three non orthogonal directions of information, graphs are essentially a projection of this information onto a two dimensional paradigm, hence the number of fundamentally equivalent ways of expressing the relation. The choice between them is based entirely on which way the information is to be accessed.

Intuitively a link is created from a to b if there is a rule that has a or $\mathrm{not}\ a$ in the body and b in the head. Nodes are annotated with atoms and atoms will be used interchangeably with 'the node representing the atom'. This means that for every rule r there will be $|B(r)|$ links to $H(r)$. As all self referential rules have been removed no node will link directly to itself. For these to be of any use, there have to be ways of extracting information from them.

Definition 7. *Let $D = (N, L)$ be the dependency graph of program P and let $a \in \mathcal{B}_P$. The reachability set after n steps from atom a is notated as $R^n(a)$ and defined as:*

$$R^1(a) = \{b \in \mathcal{B}_P | \exists (n_a, n_b, r) \in L\}$$
$$R^n(a) = \{b \in R^1(c) | c \in R^{n-1}(a)\} \cup R^{n-1}(a)$$

This is the obvious definition, the set of nodes which are reachable after travelling along n links. The following propositions give basic properties of the reachability function and link it to the concept of implication.

Proposition 3. *Let P be a program, D its dependency graph and a an atom then there exists $f_a \in \mathbb{N}$ s.t. $R^{f_a}(a) = R^{f_a+1}(a)$*

For clarity some simple notation is introduced. In the context of the preceding proposition $R^{f_a}(a)$ is notated as $R^\omega(a)$. If $b \in R^1(a)$, b is said to be directly reachable from a, while if $b \in R^\omega(a)$ it is just said to be reachable from a.

Proposition 4. *Let P be a program, $D = (N, L)$ it's dependency graph and $a \in \mathcal{B}_P$, $A \subset \mathcal{B}_P$ then: $I(+, P, A, +a) \subset R^1(a)$, $I(+, P, A, -a) \subset R^1(a)$,*
$$I(-, P, A, +a) \subset R^1(a), I(-, P, A, -a) \subset R^1(a)$$

Thus all of the implications of altering atom a and all of their knock on implications will be contained within R^ω, giving a technique for bounding the possible changes caused by adding a rule to a program. To use this and to infer that atoms cannot be altered and thus their status propagated to new answer sets results that relate the concept of reachability to the answer sets of the corresponding program are needed.

Definition 8. *Let P be a program and $a, b \in \mathcal{B}_P$. Then a may effect the status of b in an answer set $\Leftrightarrow$ there exists a chain of rules $(r_1, r_2, \ldots, r_n)$ with $r_1, \ldots, r_n \in P$ such that $(a \in B(r_1)) \vee (\mathrm{not}\ a \in B(r_1))$, $(H(r_n) \in B(r_{n+1})) \vee (\mathrm{not}\ H(r_n) \in B(r_{n+1}))$ and $H(r_n) = b$.*

Although comple, this definition is conceptual simple. If there are rules $(r_1, \ldots, r_n)$ such that a can effect whether r_1 is applicable, r_n can effect whether b appears in an answer set and each rule can influence the next then it is fair to say that a can influence whether b is in an answer set. After applying the reduct, this chain may allow the immediate consequence operator to conclude b if the status of a is known. This allows the concepts of the dependency graph to be related to answer sets.

Theorem 1. *Let P be a program and $a, b \in \mathcal{B}_P$.*

$$b \in R^\omega(a) \Leftrightarrow a \textit{ may effect the status of } b$$
$$b \notin R^\omega(a) \Leftrightarrow \neg(a \textit{ may effect the status of } b)$$

Thus there exists a link between the maximum implications of a change and the reachability set from the changed atom. Also shown is the equivalence of being reachable and being able to effect another atoms existence in an answer set.

4 The Algorithm

The first algorithm to be considered is that which handles updating the answer sets of a program after the addition of a new rule. It is divided into three phases on the basis of what depth of analysis is being performed. The first phase works on all of the answer sets and essentially answers the question "Where can the new rule be used?". Answer sets in which the rule can be used are then passed on to phase two. If there are no such answer sets then the algorithm terminates. Phase two answers the question "What *might* this rule change?". For each answer set if there are things that could possibly be changed then it is passed on to the third phase, if not then the processing for that particular answer set is complete. Finally the third phase answers the question "What *does* this rule change?" by working through and resolving the actual changes caused. This may eliminate the individual answer set, alter it or create several new answer sets from it.

Theorem 2. *Given a program P with answer sets $a_1, \ldots a_n$ and a rule r, the described algorithm is sound and complete with respect to the answer set semantics and terminates in a finite amount of time.*

4.1 Phase One

If the answers sets of a program are viewed in the model theoretical characterisation then it is clear that adding a rule r to an answer set S will not change anything if r is either applied or not applicable with respect to S. In the former case the rule will not alter the eventual conclusion of the direct consequence operator. The latter case implies that either the rule will be removed by the reduct or will contain unsupported atoms. Only if the rule is applicable but not applied does it stop S being an answer set. Phase one of the algorithm reduces the problem to cases where r is applicable but not applied and handles some edge cases (i.e. the introduction of new atoms to the system).

Description The first and most important distinction that has to be made is whether the existing program has any answer sets or not. If it does not, in some sense there is no information to work with. There are a few criteria that can be applied; the rule must allow the alteration of existing information (i.e. it is not a constraint and the head is used in the body of another rule) and all of the positive atoms in the body have non empty support (this will remove positive dependency on atoms that are new to the system). Apart from these there is nothing that can be done to resolve cases without completely recomputing the answer sets.

On the other hand, if there are answer sets already the algorithm has something to work with. Constraints are handled first as they can only remove answer sets. If the rule being added is a constraint it is simply a case of removing all answer sets with respect to which it is applicable. Handling other types of rules is more complex as they may lead to implications. First the list of answer sets is split into three lists. One list contains all of the answer sets with respect to which the new rule is not applicable, one for applied, and one for applicable but not applied. These are labelled A, B and C respectively. As

previously noted answer sets in lists A and B are unchanged and become answer sets of the new program. If list C is empty then the first part of the addition is complete. If B is non empty we already have a list of the candidates that every answer set in C will become, thus answer sets in C are removed and again the first part of this phase is complete. The final case is when both B and C are non-empty in which case the list C and a reference to the rule being added must be passed to phase two of the algorithm to detect and handle any possible implications rising from adding the head of the rule to the answer sets.

Finally, if the body of the rule contains any negative atoms that appear in any of the old answer sets it is possible that adding this rule during the computation of that answer set could have created a scenario in which this rule is applicable. Thus a search for answer sets in which the new rule is applied must be made. Without this it is impossible to add exclusive choices to the system. For example, the program $\{a \leftarrow \text{not } b.; b \leftarrow \text{not } a.\}$ has two answer sets $\{a\}$ and $\{b\}$. Adding the first rule to an empty rule set creates[5] a single answer set $\{a\}$, but without this additional search the second answer set will not be created when the second rule is added.

The analysis performed at this stage can be seen as determining how relevant the rule being added is.

Pseudo Code As well as the written description of each algorithm, pseudo code has been included to help make the flow of control within each phase of the algorithm more readily apparent. Each atom is regarded as a positive integer value.

```
/* Global variables */
P     // set of rules
old   // set of answer sets of P
/* Arguments */
r     // rule to be added
/* Local Variables */
A     // potential new answer sets where r is not applicable
B     // potential new answer sets where r is applied
C     // potential new answer sets where r is applicable but not applied
/* Return values */
new   // set of answer sets of P U {r}

/* Functions */
search(atoms,rules)
      // Uses a conventional solver to find answer sets of rules given atoms

/* Start of phase one */

if (old.size == 0) {
    /* Check to see if the rule can alter the existing information */
    if ((r.type != CONSTRAINT) &&
         (usage(r.head,P).size > 0)) {
        /* Check to see if it is possible to use this rule */
        for b in r.body {
            if ((b.positive == true) &&
                 (support(b,P).size == 0)) {
                return;
            }
        }
        /* Have to look for possible solutions */
        new = search(r.head U r.body, P U {r});
```

[5] An empty rule set is assumed to have a single, empty answer set

```
        return;
    } else {
        /* Cannot change the contradictory nature of the rule set */
        return;
    }
} else {
    /* There are answer sets */

    /* Handle constraints */
    if (r.type == CONSTRAINT) {
        for s in old {
            if (r.applicable(s) == true) {
                /* Drop this answer set */
            } else {
                new.add(s);
            }
        }
        return;
    }
    /* Work out the cases involved */
    for s in old {
        if (r.applicable(s) == true) {
            if (r.applied(s) == true) {
                B.add(s);
            } else {
                C.add(s);
            }
        } else {
            A.add(s);
        }
    }
    /* If r is not applicable or applied then the answer sets are maintained */
    new = A U B;
    /* Handle implications */
    if (C.size != 0) {
        new = new U phaseTwo(C,r,+r.head);
    }
    /* Search for any extra answer sets */
    for b in r.body {
        if ((b.positive == false) &&
             (support(b,P).size > 0)) {
            search(r.head U r.body, P U {r});
            break;
        }
    }
    return;
}

/* End of phase one */
```

4.2 Phase Two

Phase two of the algorithm focuses on cases when the rule to be added is applicable but not applied with respect to a subset of the answer sets of the program. Therefor each of these answer sets needs to be updated with at least the head of the this rule. Adding this atom may cause additional changes. This stage aims to resolve this in simple cases or to bound the maximum effect of them before handing off to phase three.

Description The first step of the algorithm is to check to see if there are any implications. A trivial sub case of this is when the atom given by the head of the rule is not used elsewhere in the program (for example if it is new to the system). In these cases the

atom given by the head can simply be added to the answer set. If this handles all of the answer sets in the input list then the algorithm is finished. At this stage the remaining answer sets can be generated by a search (using the previously outlined theory) using a conventional solver. Alternatively the set of atoms reachable from the head is calculated. By theorem 1 anything outside this cannot be changed by the addition of the rule. The information could then be passed to a conventional solver or used to construct a subset of the reachability graph in which only 'unknown' (i.e. reachable) nodes and the links between them are present. A copy of this - along with the rule being added and the answer set in question is passed to the third phase of the algorithm. It is worth noting that the body of the rule as well as the value of all unreachable nodes is passed to phase three. If the head of the rule can effect it's own body (i.e. exclusive choice) then if the body is not assumed to be true, phase three could waste time generating answer sets in which the new rule is not applicable (all of which are already known).

Phase one of the algorithm essentially tests the relevance of the rule being added, i.e. can it be used to add information to the current context. Phase two is then testing the significance of the information added; how much does it change the context it is used in.

Pseudo Code

```
/* Global Variables */
P    // The program (before addition or after subtraction)
D    // The dependancy graph of program P
/* Arguments */
S       // The set of answer sets to be altered.
r       // The rule to be added / removed
change  // The alteration to be made to the answer sets
/* Local Variables */
reachable // The set of atoms reachable from H(r)
sub       // A subgraph of D
/* Return Value */
new  // The answer sets of the augmented program

/* Functions */
implications(positive,ruleSet, atomSet, change)
    // Implications of the given change in atom set and the given rules
generateReachability(atom,graph)
    // Generate the set of things reachable from atom in graph
generateSubGraph(atomSet,graph)
    // Generates the subgraph of graph containing only nodes in atomSet
augmentSubGraph(answerSet,graph)
    // Labels each node of the graph with the value it takes in answerSet

/* Start of phase two */

/* First check to see if any of the answer sets have trivial implications */
for a in S {

    if ((implications(true,P,a,change).size() == 0) &&
         (implications(false,P,a,change).size() == 0)) {
        /* Then the changes do not progress further *
           and a U r.head is an answer set          */
        a.applyChange(change);
        new.add(a);
        S.remove(a);
    }
}

/* Check to see if there are any case unhandled */
```

```
if (S.size() == 0) {
    return;
}
/* The changes in the remainder of S can be handled *
 * with an conventional solver as described before  */
reachable = generateReachability(r.head,D);
sub = generateSubGraph(reachable,D);
/* Handle the rest of the potential answer sets with phase three */
for a in S {
    /* Augment the sub graph with the values each node held previously */
    augmentSubGraph(a,subGraph);
    new = new U phaseThree((a-(reachable U neg(reachable))) U
                r.body U (r.head) ,graph,P,0);
}
return;

/* End of phase two */
```

4.3 Phase Three

Phase three of the algorithm resolves the implications and dependency graph that have been discovered in phase two in order to produce the answer sets of the modified program. It uses a modified version of the standard bound / reduce / branch approach. When atoms within the program are shown to be unmodified by the changes caused by adding rule r, it dynamically cuts the graph of atoms that need to be resolved, thus reducing the size of the problem further. Sections of the graph that become unreachable from any of the nodes that can still be changed take the same value they had in the original answer set; there is no longer any way that they could be anything else.

Description This section is based on a standard branch and bound algorithm. The known set is augmented until all rules are either applied or not applicable with respect to it. This is handled by looping through the rule sets, removing rules that are applied and not applicable. If head of a rule is true but not applicable, the known set is checked for the negative version of the rule head (i.e. it is known that the rule's head is not in the known set), if so a contradiction has been found and nothing is returned. Alternatively the head of the rule is added and any changes need to be resolved. At the end of each run through the rule set, any atoms that have no supporting rules left in the set of undecided rules are set to be negative (i.e. they do not appear in the answer set). Finally, after all changes have been made, if there are still any rules left in the list then the algorithm branches.

The main difference from a standard branch and bound algorithm is the use of the augmented dependency graph. When the value of an atom is decided, the resolve function is called. It removes the node and all links from it. If this is the last link to any node, i.e. there is only one rule left that dictates the status of that node and it is entirely dependent on the current node, then that node will have the same relation to its old value that the current node does. For example if the current node takes the same value that it did in the old answer set and this is the only thing required to determine another node, then that node will also behave as before. This allows a more efficient method of determining the value of nodes; sections of the graph that are cannot be affected by any of the changes to the answer set are removed and set to their previous value.

Pseudo Code

```
/* Global Variables */
P    // The program (before addition or after subtraction)
/* Arguments */
known    // A set of atoms with known values
graph    // An augmented graph s.t.
           // 1. {atoms in graph} U positive{known} U negative{known} = H{P}
           // 2. the rule corresponding to every link is in rules
rules    // A list of rules with unknown status with respect to known
rules_a  // A list of rules that are applied with respect to known
/* Local Variables */
changes     // A simple boolean flag of whether there  have been any changes
            // on one pass of the algorithm
supported   // A list of pairs of atom and binary flag, flag is set to true
            // if there are still rules that support this atom
tmp         // A temporary atom that is used in branching
known_copy  // A copy of known, used for branching
graph_copy  // A copy of the graph, used for branching
/* Return Value */
newAns   // The new answer set

/* Functions */
checkSupport(set,ruleSet) // Checks that every atom in set and no others
                          // are supported by the rules in ruleSet

/* Start phase three */

/* Work until all rules have been accounted for */
while (rules.size > 0) {
    do {
        /* Note that nothing has changed so far */
        changes = false;
        /* Clear supported flags */
        for s in supported {
            supported.clearFlag(s);
        }
        /* Attempt to resolve the status of as many  rules as possible */
        for r in rules_u {

            /* Work out the status of each rule with *
             * respect to the known set of atoms     */
            switch (r.status(known)) {
                applied : rules.remove(r);
                    rules_a.add(r);
                    break;
                applicable : rules.remove(r);
                    /* Check for contradictions */
                    if ((r.type() == CONTRADICTION) ||
                        (known.contains(-r.head)) == true) {
                        return;
                    } else {
                        resolve(r.head,+1,&known,&graph);
                    }
                    changed == true;
                    rules_a.add(r);
                    supported.remove(supported.lookupByAtom(r.head));
                    break;
                not_applicable : rules.remove(r);
                    break;
                unknown : supported.setFlag(
                        supported.lookupByAtom(r.head));
                    break;
            }
        }
        /* Set every unsupported atom to -1 */
        for s in supported {
            if (s.flag == false) {
```

```
                resolve(s.atom,-1,&known,&graph);
                changed == true;
            }
        }
        resolve(r.head,-1,&known,&graph);
    } while (changes == true);
    /* Branch */
    tmp = graph.first();
    known_copy = known;
    graph_copy = graph;
    resolve(tmp,+1,&known,&graph);
    resolve(tmp,-1,&known_copy,&graph_copy);
    phaseThree(known_copy,graph_copy,rules,rules_a);
}
if (checkSupport(known,rules_a) == true) {
newAns = known;
}
/* End of phase three */

// resolves the dependency graph
resolve(atom,value,atomSet,graph) {
    /* Update the atom set */
    atomSet.add(value);
    /* Remove all of the links from the node */
    for l in graph.lookup(atom).linksFrom {
        graph.removeLink(l);
        /* If that was the last link to the node */
        if (graph.lookup(l.target).linksTo.size == 0) {
            /* It's value must have the relation  *
             * to it's old one that this one does */
            if (graph.lookup(atom).oldValue == value) {
                resolve(l.target,l.target.oldValue,
                                atomSet,graph);
            } else {
                resolve(l.target,-l.target.oldValue,
                                atomSet,graph);
            }
        }
    }
    graph.removeNode(atom);
}
```

5 Future Research

This paper presents a complete solution for handling the addition of a single rule to a programs. However in terms of the overall topic this is only the beginning.

Subtraction of a rule should follow more or less the same pattern as the addition as the operations complement each other. The key difference is that rather than searching for answer sets in which the rule to be added is applicable but not applied, the phase one of the subtraction algorithm searches for answer sets in which the rule to be removed is the only rule with that head atom that is applied.

As this work treats changes in rule sets as single, atomic operations no consideration of the pattern of rule modification has been made. Care must be taken to add constraints before the rules that generate large numbers of options. One area of interest is to look at ways of adding multiple rules simultaneously, allowing a new concept or block of data to be added in one operation with considerable potential savings. An alternative approach to the same problem would be to develop some form of criterion or heuristic for when to handle a series of changes using the presented algorithm and when it is more efficient to recompute completely.

Such issues lead naturally to considering a modular approach to answer set and logic programming. Modular programming is a well accepted technique and a powerful abstraction mechanism, using modified versions of the presented algorithms and techniques for making several simultaneous changes it may well be possible to provide this for answer set programming. This raises more possibilities, from pre-computation of fixed blocks of rules to distributed computation to mixing rule sources (for example using databases as sources of facts) to the possibilities of choosing logical paradigms on a module by module basis (using preference based choice formalisms such as OCLP for human interaction and $AnsProlog^*$ for the actual computation). All of these would be further step towards providing a modern programming environment for answer set computation.

Finally nothing has been presented on the addition of rules in any of the formalisms that extend $AnsDatalog^{\neg,\bot}$. Those that can be mapped or reduced to $AnsDatalog$ could be converted quite easily, although the nature of such mappings may significantly reduce the value of such algorithms. However logic systems such as $AnsProlog^{or}$ with a higher computational complexity and logic system which include function symbols are a much more interesting issues. Clearly the presented algorithms provide part of, but not a complete solution.

Currently we are working on a prototype implementation of the presented algorithm. Although it is too early to proper benchmarking, the results look hopefulf, especially since the answer set solver behind is not equiped with all the heuristic one can find in establised answer set solvers.

References

1. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3-4):365–386, 1991.
2. M. J. Brain. Undergraduate dissertation: Incremental answer set programming. Technical Report 2004–05, University of Bath, U.K., Bath, May 2004.
3. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of fifth logic programming symposium*, pages 1070–1080. MIT PRESS, 1988.
4. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge Press, 2003.
5. noMoRe main web page, http://www.cs.uni-potsdam.de/ linke/nomore/
6. Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR system `dlv`: Progress report, comparisons and benchmarks. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *KR'98: Principles of Knowledge Representation and Reasoning*, pages 406–417. Morgan Kaufmann, San Francisco, California, 1998.
7. I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal LP. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programing and Nonmonotonic Reasoning*, volume 1265 of *LNAI*, pages 420–429, Berlin, July 28–31 1997. Springer.
8. Smodels main web page, http://www.tcs.hut.fi/Software/smodels/
9. DLV main web page, http://www.dlvsystem.com/