Grounding Bound Founded Answer Set Programs

Rehan Abdul Aziz and Geoffrey Chu and Peter James Stuckey
University of Melbourne and NICTA

Abstract

Bound Founded Answer Set Programming (BFASP) is an extension of Answer Set Programming (ASP) that extends stable model semantics to numeric variables. While the theory of BFASP is defined on ground rules in practice BFASP programs are written as complex non-ground expressions. Flattening of BFASP is a technique used to simplify arbitrary expressions of the language to a small and well defined set of primitive expressions. In this paper, we first show how we can flatten arbitrary BFASP rule expressions, to give equivalent BFASP programs. Next, we extend the bottom-up grounding technique and magic set transformation used by ASP to BFASP programs. Our implementation shows that for BFASP problems, these techniques can significantly reduce the ground program size, and improve subsequent solving.

Introduction

Many problems in the areas of planning or reasoning can be efficiently expressed using Answer Set Programming (ASP) (Baral 2003). Answer Set Programming enforces stable model semantics (Gelfond and Lifschitz 1988) on the program, which disallow solutions representing circular reasoning. For example, given only rules that says: b can be inferred from a, and a can be inferred from b, the assignment $a = true, b = true$ would be a solution under the logical semantics normally used by Boolean Satisfiability (SAT) (Mitchell 2005) solvers or Constraint Programming (CP) (Marriott and Stuckey 1998) solvers, but would not be a solution under the stable model semantics used by ASP solvers. Thus ASP is particularly useful in problem domains where circular reasoning needs to be avoided.

Bound Founded Answer Set Programming (BFASP) (Aziz, Chu, and Stuckey 2013) is an extension of ASP to allow founded integer and real variables. This allows us to concisely express and efficently solve problems involving inductive definitions of numeric variables where we want to disallow circular reasoning. As an example consider the Road Construction problem (*RoadCon*). We wish to decide which roads to build such that the shortest paths between various cities are acceptable, with the minimal total cost. This can be modelled as:

$$\text{minimize} \sum_{e \in Edge} built[e] \times cost[e]$$
$$\forall y \in Node : sp[y, y] \leq 0$$
$$\forall y \in Node, e \in Edge : sp[from[e], y] \leq len[e] + sp[to[e], y] \leftarrow built[e]$$
$$\forall y \in Node, e \in Edge : sp[to[e], y] \leq len[e] + sp[from[e], y] \leftarrow built[e]$$

$$\forall p \in Demand : sp[d\_from[p], d\_to[p]] \leq demand[p]$$

The decisions are which edges e are built ($built[e]$). The aim is to minimize the total cost of the edges $cost[e]$ built. The first rule is a base case that says that shortest path from a node to itself is 0. The second constraint defines the shortest path $sp[x, y]$ from x to y: the path from x to y is no longer than from x to z along edge e if it is built plus the shortest path from z to y; and the third constraint is similar for the other direction of the edge. The last constraint ensures that the shortest path for each of a given set of paths $p \in Demand$ are no longer than their maximal allowed distance $demand[p]$.

As shown above the model has a trivial solution with cost 0 by setting $sp[x, y] = 0$ for all x, y. In order to avoid this we require that the sp variables are (upper-bound) *founded* variables, that is they take the largest possible justified value. The first three constraints are actually *rules* which justify upper bounds on sp, the last constraint it a restriction that needs to be met and cannot be used to justify upper bounds. Solving such a BFASP is challenging, mapping to CP models leads to inefficient solving, and hence we need a BFASP solver which can reason directly about *unfounded sets* of numeric assumptions, see (Aziz, Chu, and Stuckey 2013) for details.

The Road Construction problem is a *non-ground* BFASP since it is parametric in the data: $Node$, $Edge$, $Demand$, $cost$, $from$, to, len, d_from, d_to and $demand$. In this paper we consider how to efficiently create a ground BFASP from a non-ground BFASP given the data. This is analogous to *flattening* (Stuckey and Tack 2013) of constraint models and *grounding* (Syrjnen 2009; Gebser, Schaub, and Thiele 2007; Perri et al. 2007) of ASP programs. The contributions of this paper are:

- A flattening algorithm that transforms complex expressions to primitive forms while preserving the stable model

semantics.

- A generalization of bottom-up grounding for normal logic programs to BFASPs.
- A generalization of the magic set transformation for normal logic programs to BFASPs.

Preliminaries

Constraints

We consider three types of variables: integer, real, and Boolean. Let $\mathcal{V}$ be a set of variables. We use $[l, u]$ to indicate the interval $l \leq x \leq u$. A *domain* D maps each variable $x \in \mathcal{V}$ to a set of constant values $D(x)$. A *valuation* (or assignment) θ over variables $vars(\theta) \subseteq \mathcal{V}$ maps each variable $x \in vars(\theta)$ to a value $\theta(x)$. A restriction of assignment θ to variables V, $\theta|_V$, is the the assignment θ' over $V \cap vars(\theta)$ where $\theta'(v) = \theta(v)$.

A *constraint* c is a set of assignments over the variables $vars(c)$, representing the solutions of the constraint. Given a constraint c, a variable $y \in vars(c)$ is *monotonically increasing (decreasing)* in c if forall solutions $\theta \in c$, then increasing (decreasing) the value of y also creates a solution, that is θ' where $\theta'(y) > \theta(y)$, and $\theta'(x) = \theta(x), x \in vars(c) - \{y\}$, is also a solution of c.

A *constraint program (CP)* is a collection of variables $\mathcal{V}$ and constraints C on those variables ($vars(c) \subseteq \mathcal{V}, c \in C$). A *positive-CP* P is a CP where each constraint is increasing in exactly one variable and decreasing in the rest. The *minimal* solution of a positive-CP is an assignment θ that satisfies P s.t. there is no other assignment θ' that also satisfies P and there exists a variable v for which $\theta'(v) < \theta(v)$. Note that for Booleans, $true > false$. A positive-CP P always has a unique minimal solution. This unique minimal solution θ is given by: $\forall x, \theta(x) = min\{v \mid P \Rightarrow x \geq v\}$. If we have bounds consistent propagators for all the constraints in the program, then the unique minimal solution can be found simply by performing bounds propagation on all constraints until a fixed point is reached, and then setting all variables to their lowest values.

Answer set programming

A *normal logic program* P is a collection of *rules* of the form:

$$b_0 \leftarrow b_1 \wedge \ldots \wedge b_n \wedge \neg b'_1 \wedge \ldots \wedge \neg b'_m$$

where $\{b_0, b_1, \ldots, b_n, b'_1, \ldots, b'_m\}$ are Boolean variables. b_0 is the *head* of the rule while the RHS of the reverse implication if the *body* of the rule. A rule without any negative literals is a *positive rule*. A *positive program* is a collection of positive rules. The *least model* of a positive program is an assignment θ that assigns $true$ to the minimum number of variables. The *reduct* of P w.r.t. an assignment θ is written P^θ is a positive program obtained by removing the negative literals $\{b'_1, \ldots, b'_m\}$ in every rule r of P as follows: if there exists an i for which $\theta(b'_i) = true$, discard the rule, otherwise, discard all the negative literals from the rule. The stable models of P are all assignments θ for which the least model of P^θ is equal to θ. Note that if we consider a logic program as a constraint program, then a positive program is a positive-CP and the least model of that program is equivalent to the minimal solution defined above.

Bound Founded Answer Set Programs (BFASP)

BFASP is an extension of ASP that extends its semantics over integer and real variables. In BFASP, the set of variables is a union of two disjoint sets: standard $\mathcal{S}$ and *founded* variables $\mathcal{F}$.[1] A rule r is a pair (c, y) where c is a constraint, $y \in \mathcal{F}$ is the head of the rule and it is increasing in c. A bound founded answer set program (BFASP) P is a tuple $(\mathcal{S}, \mathcal{F}, C, R)$ where C and R are sets of constraints and rules respectively. Given a variable $y \in \mathcal{F}$, $rules(y)$ is the set of rules with y as their heads. Each standard variable s is associated with a lower and an upper bound, written $lb(s)$ and $ub(s)$ respectively. On the other hand, founded variables are only associated with lower bounds that we refer to as *initial bounds*, that can be accessed through the function ujb. An initial bound b for a numeric founded variable y can be considered as an implicit rule $(y \geq b, y)$. For Booleans, we assume that the initial bounds are fixed to $false$. Note that we consider the domain of Boolean variables to be ordered such that $true > false$. So for example, an ASP rule such as $a \leftarrow b \wedge c$ can equivalently be written as: $a \geq f(b, c)$ where f is a Boolean function which returns the value of $b \wedge c$.

The reduct of a BFASP P w.r.t. an assignment θ is a positive-CP made from each rule $r \equiv (c, y)$ by replacing in c each variable $x \in vars(c) - \{y\}$ that is not decreasing by its value $\theta(x)$ to create a positive-CP constraint c'. Let r^θ denote this constraint. If r^θ is not a tautology, it is included in the reduct. An assignment θ is a stable model of P iff i) it satisfies all the constraints in P and ii) it is the minimal model that satisfies P^θ and all the initial bounds on founded variables

Example 1. *Consider a BFASP with standard variable s, integer founded variables a, b both with initial bounds of 0, Boolean founded variables x and y, and the rules:*

$$(a \geq b + s, a)$$
$$(b \geq 8 \leftarrow x, b)$$
$$(x \leftarrow \neg y \wedge (a \geq 5), x)$$

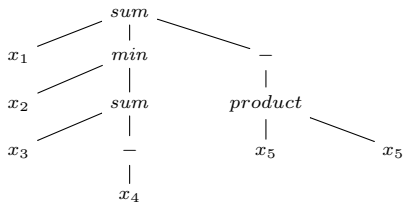
Consider an assignment θ s.t. $\theta(x) = true$, $\theta(y) = false$, $\theta(b) = 8$, $\theta(s) = 9$ and $\theta(a) = 17$. The reduct of θ is the positive-CP: $a \geq b + 9$, $b \geq 8 \leftarrow x$, $x \leftarrow a \geq 5$. The minimal model that satisfies the reduct and the initial bounds is equal to θ, therefore, θ is a stable model of the program. Consider another assignment θ' where all values are the same as in θ, but $\theta'(s) = 3$. Then, $P^{\theta'}$ is the positive-CP: $a \geq b + 3$, $b \geq 8 \leftarrow x$, $x \leftarrow a \geq 5$. The minimal solution that satisfies this positive-CP and all the initial bounds is M where $M(a) = 3$, $M(b) = 0$, $M(x) = M(y) = false$.

The focus of this paper is BFASPs where every rule is written in the form $(y \geq f(x_1, \ldots, x_n), y)$. $f(x_1, \ldots, x_n)$

[1] For the rest of this paper we only consider *lower bound* founded variables, analogous to founded Booleans. Upper bound founded variables can be implemented as negated lower bound founded variables, e.g. replace $sp[x, y]$ in the Road Construction example by $-nsp[x, y]$ where $nsp[x, y]$ is lower bound founded.

is essentially an expression tree where the leaf nodes are the variables $x_1, \ldots, x_n$. A variable is *terminal* in an expression if it appears as a direct descendant of the root.

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



The *local dependency graph* for a BFASP P is defined over founded variables. For each rule $r = (y \geq f(x_1, \ldots, x_n), y)$, there is an edge from y to all founded x_i. Each edge is marked increasing, decreasing, or non-monotonic, depending on whether f is increasing, decreasing, or non-monotonic in x_i. A BFASP is *locally valid* iff no edge within an SCC is marked non-monotonic. A program is *locally stratified* if all the edges between any two nodes in the same component are positive. For example, if x and y are in the same SCC, then $y \geq sin(x_1)$ where x_1 has initial domain $(-\infty, \infty)$ is not valid since the sin function is not monotonic over this domain, but $y \geq sin(x_1)$ where x_1 has initial domain $[0, \pi/2]$ is valid.

We say $\theta \in D$ if $\theta(x) \in D(x), \forall x \in \mathcal{V}$.

Non-ground BFASPs

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

Example 3. *Suppose we have a 2 dimensional array of variables* a *and 1 dimensional arrays of variables* b *and* c*. In the non-ground rule:* $\forall i, j \in [1, 10]$ *where* $i < j : (a[i, j] \geq b[i + 1] + c[i * j], a[i, j])$*,* i *and* j *are the index variables,* $i, j \in [1, 10] \wedge i < j$ *is the generator constraint,* $l_0(i, j) = (i, j)$*,* $l_1(i, j) = (i + 1)$*,* $l_2(i, j) = (i * j)$*, and* $f(x_1, x_2) = x_1 + x_2$*.*

Note that we require the generator constraint in each rule to constrain the index variables so that f is always defined. For example, if we have variables $a[1], \ldots, a[10]$, $b[1], \ldots, b[10]$ and $c[1], \ldots, c[10]$, then the rule $\forall i \in [1, 10](a[i] \geq b[i + 1] + c[i + 2], a[i])$ is not valid since $c[i + 2]$ refers to a variable outside the array when $i = 10$. On the other hand, $\forall i \in [1, 8] : (a[i] \geq b[i+1]+c[i+2], a[i])$ is valid since all values of i refer to variables within the array. We can relax this restriction but it requires us to carefully treat partial function applications (see e.g. (Frisch and Stuckey 2009)).

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

Given a non-ground rule r, let $grnd(r)$ be the set of ground rules obtained by substituting all possible values of the index variables that satisfy $gen(r)$ into the quantified expression. Similarly given a non-ground BFASP P, let $grnd(P)$ be the grounded BFASP that contains the grounding of all its rules and constraints.

The *predicate dependency graph*, validity and stratification are defined similarly for array variables and non-ground rules as the local dependency graph, local validity and local stratification respectively are defined for atomic variables and ground rules. All our subsequent discussion on non-ground BFASPs is restricted to valid BFASPs. Note that similarly to how local stratification is a tighter condition for logic programs, local validity is also a tighter condition for BFASPs. In particular, non-ground rules where the monotonicity of subexpressions are not fixed may still be locally valid.

Flattening

A ground BFASP may contain constraints and rules whose expressions are not *flat*, i.e., they are expression trees with height greater than one. Such expressions are not supported by constraint solvers and we need to flatten these expressions to primitive forms. We omit consideration of flattening constraints since this is the same as in standard CP (Stuckey and Tack 2013). Consider the expression tree in Example 2, if it were a constraint, we would introduce variables $i_1, \ldots, i_5$ to decompose the given function into the following set of equalities: $f = x_1 + i_1 + i_2, i_1 = min(x_2, i_3), i_3 = x_3 + i_4, i_4 = -x_4, i_2 = -i_5, i_5 = x_5 \times x_5$. Similar to how an arithmetic function is decomposed by introducing new variables and a set of equalities, rules in BFASP can be flattened by introducing new founded and standard variables, and new rules and constraints. In this section, we show how this can be done such that the stable models of the original program are preserved.

We first note that the standard CP flattening algorithm does *not* preserve stable model semantics. If a standard variable is introduced in order to represent a subexpression containing founded variables, the stable models of the program may change.

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

To preserve the stable model semantics, it is in fact necessary to use introduced *founded* variables to represent subexpressions containing founded variables. Before we describe the central result that we use in our flattening algorithm, let us describe two restrictions that we impose on the rules that we require for the correctness of our algorithm:

1. Each founded variable appears only once in the expression tree of the rule expression. For example, the rule $y \geq x_1 - min(x_1, x_2)$ is not supported since x_1 appears twice.
2. Every function in which a founded variable appears must be monotonic in all its argument given the initial domains of the variables.

These restrictions force the BFASP to be locally valid. The following theorem shows us how we can preserve the stable model semantics.

Theorem 1. *Let* P *be a BFASP containing a rule* $r \equiv (y \geq f_1(x_1, \ldots, x_k, f_2(x_{k+1}, \ldots, x_n)), y)$ *where* f_1 *is increasing in the argument where* f_2 *appears. Let* P' *be the same as* P *with* r *replaced by the two rules:* $r_1 \equiv (y \geq f_1(x_1, \ldots, x_k, y'), y)$ *and* $r_2 \equiv (y' \geq f_2(x_k, \ldots, x_n), y')$ *where* y' *is an introduced lb-founded variable. Then the stable models of* P' *restricted to the variables of* P *are equivalent to the stable models of* P.

Theorem 1 tells us that we can preserve the stable model semantics by introducing a founded variable to represent subexpressions containing founded variables. Note that if a subexpression does not contain any founded variables at all, i.e., only contains standard variables, parameters or constants, then a standard CP flattening step is sufficient.

Example 5. *Consider the rule:* $(y \geq x_1 + min(x_2, x_3 - x_4) - (x_5)^2, y)$. *Suppose* x_1, x_2, x_5 *are lb-founded variables, and* x_3, x_4 *are standard variables. The RHS of the initial rule is increasing in the subexpression* $min(x_2, x_3 - x_4)$ *and decreasing in* $(x_5)^2$, *so we can replace it with founded variables* i_1 *and* i_2, *and break it into:* $(y \geq x_1 + i_1 + i_2, y)$, $(i_1 \geq min(x_2, x_3 - x_4), i_1)$, $(i_2 \geq -(x_5)^2, i_2)$. *The rule* $(i_1 \geq min(x_2, x_3 - x_4), i_1)$ *requires further flattening. However, the subexpression* $x_3 - x_4$ *only contains standard variables, so we only need to introduce a standard variable* i_3 *and break it into:* $(i_1 \geq min(x_2, i_3), i_1)$ *and* $i_3 = x_3 - x_4$.

The flattening algorithm, formalized as the procedure flat, works as follows. We put all the rules of the program in a set R and all the constraints in the program in a set T. We pick a rule $r \equiv (y \geq f(e_1, \ldots, e_n), y)$ from R, where f is the top level function in that rule, and $e_1, \ldots, e_n$ are the expressions which form f's arguments. If there is some e_i which is not a terminal, i.e., not a constant, parameter or variable, then we have two cases. If e_i is an expression that does not contain any founded variables, we simply introduce a standard variable y', replace e_i with y' in that rule, and add the constraint $y' = e_i$ to T. If e_i is an expression that does contain founded variables, then we must apply the transformation described in Theorem 1. If f is increasing in e_i, we introduce an lb-founded variable y', replace e_i with y' in that rule, and add a new rule $(y' \geq e_i, y')$ to R. If f is decreasing in e_i, we introduce an lb-founded variable y', replace e_i with $-y'$ in that rule, and add a new rule $(y' \geq -e_i, y')$ to R. After this, all the arguments of f are terminals. Through the subroutine simplify, we simplify r as much as possible, e.g., by getting rid of double negations, pushing negations inside the expressions as much as possible etc. We remove r from R and pick another rule from R until R is empty. Finally, we flatten all the constraints in T using the standard CP flattening algorithm cp_flat as described in (Stuckey and Tack 2013). Note that since we replace all decreasing subexpressions by negated introduced variables and simplify expressions by pushing negations towards the variables, we handle negation through simple rule forms like $(y \geq -x, y)$, $(y \geq \frac{1}{x}, y)$, $(y \geq \neg x, y)$ etc.

```
flat(P)
  P_flat := ∅
  R := rules(P)
  T := cons(P)
  for(r ≡ (y ≥ f(e_1, ..., e_n), y) ∈ R)
    R := R \ {r}
    for(each non-terminal e_i)
      if(e_i does not contain founded vars)
        replace e_i with standard var y' in r
        T = T ∪ {y' = e_i}
      elif(f is increasing in e_i)
        replace e_i with lb-founded var y' in r
        R = R ∪ {(y' ≥ e_i, y')}
      elif(f is decreasing in e_i)
        replace e_i with −y' in r
        R = R ∪ {(y' ≥ −e_i, y')}
    r := simplify(r)
    P_flat = P_flat ∪ {r}
  for(c ∈ T)
    P_flat = P_flat ∪ cp_flat(c)
  return P_flat
```

Once we have flattened the entire program, we can calculate the initial domains for the introduced variables as well as the ujb values for the introduced founded variables. This

step is not strictly necessary for correctness since we can always set the ujb values of founded variables to $-\infty$ and set the initial domains of introduced variables to $(-\infty, \infty)$. However, getting tighter initial domains may allow us to get a smaller grounding and/or improve solving efficiency. We perform these calculations in the reverse order in which the variables were introduced. Finding initial domains simply requires propagating the constraints and is standard in CP flattening so we do not discuss this further. Suppose y' is an introduced lb-founded variable. There will be exactly one rule $(y' \geq f(x_1, \ldots, x_m), y)$, which means that the stable model semantics forces it to be equal to $f(x_1, \ldots, x_m)$. We set $ujb(y') = f(\theta(x_1), \ldots, \theta(x_n))$ where:

1. if x_i is a founded variable and f is increasing in x_i, then $\theta(x_i) = ujb(x_i)$
2. else if f' is increasing in x_i, then $\theta(x_i) = lb(x_i)$
3. else $\theta(x_i) = ub(x_i)$.

Example 6. *Assuming the initial domains of all variables were* $[0, 10]$*, and* $ujb(x_1) = 1$*,* $ujb(x_2) = 2$ *and* $ujb(x_5) = 5$*, then the domains of the introduced variables of Example 4 are:* $i_3 \in [-10, 10]$*,* $i_2 \in [-100, 0]$*,* $i_1 \in [-10, 10]$ *and the* ujb *values are:* $ujb(i_2) = -25$ *and* $ujb(i_1) = -10$*.*

The algorithm can be extended to non-ground rules by defining the index set of the introduced variables to be equal to the domain of local variables as given in the generator of the original rule in which their bodies appear. Moreover, the generator expression of the intermediate rules stay the same as that of the original rule from which they are derived. For initial bound calculation on the introduced variables, θ is defined similarly to how it is defined for standard variables.

Grounding

ASP grounders rely on two operations: *propagation* and *justification*. Propagation means inferring that some variable is true in all stable models of the program. For example, let us say that a is given as a fact, then any rule like $a \leftarrow b$ is *useless* since removing it does not affect the stable models of the program. Justification means inferring that a variable can be true in some stable model of the program. For example, if we learn that b can be justified by some rule, and there is another rule $a \leftarrow b$, then we must include this rule in the program since it offers a justification for a. Justification also has a potential to eliminate a useless rule, e.g., if we know that b has no rule justifying it, then we can discard the rule $a \leftarrow b$, or postpone including this in our ground program until we find a justification for b.

In constraints term, what ASP grounders do is maintain an implicit domain D for all the variables, which initially only contains $false$ for all variables. By inspecting the rules, they then iteratively increase the upper bound on these variables (from $false$ to $true$), which in turn increases the upper bound of bodies of some rules in which these variables appear as positive literals, which results in the instantiation of those rules which increases the upper bound on their heads and so on. At the same time, they also propagate the lower bound of a variable (fix it $true$ where possible). By performing both justification and propagation, the resulting ground program has the same stable models as the ground program generated by exhaustive instantiations, but has no useless rules.

To implement grounding based only on justification, due to the simple nature of normal logic rules, it is sufficient to keep track of ground variables and derive further instantiations based solely on this knowledge, without actually explicitly maintaining a domain. For example, if variables b and c have been created, then $a \leftarrow b \wedge c$ must also be created. Justification of all positive literals immediately implies justification of the head. Unfortunately, BFASP is more complicated than that. For example, if we have a rule $a \geq b$ where the initial bound of a is 5 and that of b is 1, then if some rule justifies a bound of 2 on b, this does not imply that $a \geq b$ should be grounded. Until we learn that a bound of 6 can be justified on b, the rule is useless since not including it in the final program does not change the stable models of the program. In general, in BFASP, justification higher than the initial bound for a variable does not imply usefulness of all rules in which it positively appears. This means that a fixed point bottom-up grounding based on creation of variables is not sufficient to guarantee a useless rules free ground program. However, this does not mean that it is not useful, by allowing some over grounding, we can still eliminate significant number of useless rules for certain problems. We present one such approach in this section.

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

We refer to a variable x as being *created*, written $cr(x)$, if it can go above its initial bound. More formally, $cr(x)$ is a founded Boolean with a rule: $cr(x) \leftarrow x > ujb(x)$. While that is how we define $cr(x)$, we do not explicitly have a variable $cr(x)$ and the above rule in our implementation. Instead, we implement it by maintaining a set Q of variables that have been created. Initially, Q is empty. We recursively look at each non-ground rule to see if the newly created variables make it possible for more head variables to be justified above their ujb values. If so, we create those variables and add them to Q. In order to do this, we need to find necessary conditions under which the head variable can be justified above its ujb. In order to simplify the presentation, we are going to define ujb for constants, standard variables and parameters as well. For a constant x, we define $ujb(x)$ to be the value of x. For parameters and standard variables x, we define $ujb(x) = ub(x)$.[2] Table 1 gives a non-exhaustive list

[2]Upper and lower bounds for parameters can be established by simply parsing the array.

c	ϕ_r
$y \geq sum(x_1, \ldots, x_n)$	$(\sum_i ujb(x_i) > ujb(y)) \vee ((\wedge_i ujb(x_i) > -\infty \vee cr(x_i)) \wedge (\vee_i cr(x_i)))$
$y \geq max(x_1, \ldots, x_n)$	$\vee_i (ujb(x_i) > ujb(y) \vee cr(x_i))$
$y \geq min(x_1, \ldots, x_n)$	$\wedge_i (ujb(x_i) > ujb(y) \vee cr(x_i))$
$y \geq product(x_1, \ldots, x_n)$ where $\wedge_i x_i > 0$	$\prod_i ujb(x_i) > ujb(y) \vee (\vee_i cr(x_i)))$
$y \geq x \leftarrow r$	$r \in Q \wedge (ujb(x) > ujb(y) \vee cr(x))$
$y \leftarrow x \geq 0$	$ujb(x) >= 0 \vee cr(x)$
$y \leftarrow \wedge_i x_i$	$\wedge_i cr(x_i)$
$y \leftarrow \vee_i x_i$	$\vee_i cr(x_i)$
$y \geq -x_1$	$-ub(x_i) > ujb(y)$
$y \leftarrow \neg x_1$	$true$
$y \geq 1/x_1$ where $x_1 > 0$	$1/-ub(x_1) > ujb(y)$

Table 1: *Grounding conditions for rule* $r = (c, y)$

of necessary conditions for the head variable to be justified above its ujb value for different rule forms.

Let us now make a few observations about the conditions given in Table 1. A key point is that for many rule forms ϕ_r can evaluate to $true$, even without any variable in the body getting created. For example, for max, even if one variable has an initial bound that is greater than the initial bound of the head, the rule needs to be grounded completely. All such rules that evaluate to true give us a starting point for initializing Q in our implementation.

The linear case (sum) deserves some explanation. It is made up of two clauses, the first of which is an evaluation of the initial condition, i.e., whether the sum of initial bounds of all variables is greater than the initial bound of the head. If this condition is true, then the rule needs to be grounded unconditionally. If this is false, then the second clause becomes important. The second clause itself is a conjunction of two more clauses. The first part says that all variables must be greater than $-\infty$ in order for the rule to justify a finite value on the head. In the case where all variables already have a finite initial bound, the final part of the condition says that at least one of them must be created for the rule to be grounded (given the initial condition failed). Note that this condition becomes redundant if at least one of the variables has an initial bound of $-\infty$. A final observation is that after evaluating initial bounds, all conditions given in the table simplify to one of the following four forms: $true$, $false$, $\vee_i cr(x_i)$ or $\wedge_i cr(x_i)$.

We can argue about the correctness of our approach by looking at each row in the table and reasoning that until the condition is satisfied, the rule can be ignored without affecting the stable models of the program. We only provide a brief sketch and do not analyze each case in the table. Say, e.g. for $y \geq max(x_1, \ldots, x_n)$, if the condition is not satisfied, this means that no x_i has a rule in the program that justifies a value higher than its initial bound, and no x_i initially justifies a bound on y that is greater than $ujb(y)$. If we include a ground version of this rule in the final program, then after taking the reduct w.r.t. some assignment, the rule can never justify any bound on the head during the minimal order computation, and hence we can safely eliminate it.

Example 7. *Consider a BFASP with the following two non-ground rules:*

$$\forall i \in [1, 10] : a[i] \geq b[i] + x[i]$$
$$\forall i \in [1, 10] : x[i] \geq min(c[i], d[i])$$

where x is an introduced variable. The initial bounds are as follows: $ujb(a) = 5$, $ujb(b) = 2$, $ujb(c) = 7$, $ujb(d) = 1$ and $ujb(x) = 1$. For the first rule, the initial condition evaluates to false. Moreover, since both b and x have initial bounds greater than $-\infty$, we get:

$$cr(b[i]) \vee cr(x[i])$$

For the second rule, since $ujb(c[i]) > ujb(x[i])$ and $ujb(d[i])$ is not greater than $ujb(x[i])$, we get the condition: $cr(d[i])$.

createCPs(P)
- **for**($r \in P : \phi_r = \bigwedge_{i=1}^{n} cr(x_i[\bar{l_i}])$)
  - $cp[r]$:= $true$ % new constraint program
  - $cp[r]$:= $cp[r] \wedge gen(r)$
  - **for**($i \in 1 \ldots n$)
    - $set[r, i]$:= $\emptyset$
    - $cp[r]$:= $cp[r] \wedge \bar{l_i} \in \ll set[r, i] \gg$
- **for**($r \in P : \phi_r = \bigvee_{i=1}^{n} cr(x_i[\bar{l_i}])$)
  - **for**($i \in 1 \ldots n$)
    - $cp[r, i]$:= $true$ % new constraint program
    - $cp[r, i]$:= $cp[r, i] \wedge gen(r)$
    - $set[r, i]$:= $\emptyset$
    - $cp[r, i]$:= $cp[r, i] \wedge \bar{l_i} \in \ll set[r, i] \gg$

ground(P)
- $C := \{\text{groundAll(c)} : c \in constraints(P)\}$
- $R' := \{\text{groundAll(r)} : r \in P : \phi_r = true\}$
- **while**($R' \neq \emptyset$)
  - H := heads(R')
  - $Q \cup= H$
  - R' := $\emptyset$
  - **for**($r \in P : H \cap vars(\phi_r) \neq \emptyset$)
    - **if**($\phi_r \neq \bigwedge_{i=1}^{n} cr(x_i[\bar{l_i}]) \wedge \phi_r(Q) \neq \bigwedge_{i=1}^{n} cr(x_i[\bar{l_i}])$)
      - continue
    - **for**($i \in 1 \ldots n$)
      - $dom = \{\bar{m} \mid x[\bar{m}] \in Q\}$
      - $set[r, i]$:= $dom \setminus set[r, i]$
      - **if**(ϕ_r is conj) $R' \cup=$ search($cp[r]$) $\setminus R$
      - **if**(ϕ_r is disj) $R' \cup=$ search($cp[r, i]$) $\setminus R$
      - $R \cup= R'$
      - $set[r, i]$:= cr

We are now ready to present the main bottom-up grounding algorithm. createCPs is a preprocessing step that creates constraint programs for rules in a BFASP P whose conditions are either conjunctions or disjunctions. For a rule with a conjunctive condition, it only creates one program, while for one with a disjunctive condition, it creates one constraint program for each variable in the condition. Each program is initialized with the $gen(r)$ which defines the variables and some initial constraints given in the where clause

of the non-ground rule. Furthermore, for each array literal in the ϕ_r, a constraint is posted on its literal (which is a function of index variables in the rule), to be in the domain given by the *current* value of the set variable (the reason for the Quine quotes) which is initially set to empty.

ground is called on after preprocessing. Q and R are sets of ground variables and rules respectively. groundAll is a function that grounds a non-ground rule or constraint completely, and returns the set of all rules and constraints respectively. Initially, we ground all constraints in P and rules for which ϕ_r evaluates to true. R' is a temporary variable that represents the set of new ground rules from the last iteration. In each iteration, we only look for non-ground rules that have some variable in their conditions that is created in the previous iteration. heads takes a set of ground rules as its input and returns their heads. In each iteration, through Q, we manipulate the set constraint to get new rule instantiations. For each variable in the clause, we make set equal to the new values created for that variable. For both the conjunctive and the disjunctive case, this optimization only tries out new values of recently created variables to instantiate new rules. search takes a constraint program as its input, finds all its solutions, instantiates the non-ground rule for each solution, and returns the set of these ground rules. After creating new rules due to the new values in set, we make it equal to all values of the variable in Q. The fixed point calculation stops when no new rules are created.

Magic set transformation

Let us first define the *query* of a BFASPs. To build the query Q for a BFASP P, we ground all its constraints and its objective function, and put all the variables that appear in them in Q.[3] Note that our query does not have any free variable and only contains ground variables. Therefore, we do not need any adornment strings to propagate binding information as in the original magic set technique. The original magic set technique has three stages: adorn, generate and modify. For the reason described above, we only describe the latter two.

The purpose of the magic set technique is to simulate a top-down computation through bottom-up grounding. This is achieved by creating a corresponding *magic variable* m_a for every variable a in the original program that represents whether we care about that variable or not. Additionally, there are *magic rules* that specify when a magic variable should be created, meaning, when we become interested in the corresponding variable. Consider a simple rule $(a \geq b + c + d, a)$. Let us say that the initial bounds of all four variables are $-\infty$, and we are interested in computing the final value of a. We model this as initially setting m_a to true. To capture that the value of b is required to compute the value of a, we add a magic rule $m\_b \leftarrow m\_a$. We can have a similar rule for m_c, but actually, we can make the condition for deriving m_c tighter. If b can never go higher than $-\infty$, then there is no need to know the value of c since the rule is useless until b is created. Similarly, we are only interested in d if both b and c are created.

[3]Technically if the problem has output variables, whose value will be printed, they too need to be added to Q.

Fortunately, we already have the necessary conditions in the form of ϕ_r that should be satisfied before a non-ground rule can be instantiated to a useful ground rule. We now describe how we can utilize that information for the generation of magic rules. First, recall that after evaluating the initial conditions, for any rule r, ϕ_r reduces to true, false, a conjunction or a disjunction. For a conjunction, the magic rules are the same as they are for a normal rule in the original magic set technique. For example, for the above rule $r = (a \geq b + c + d, a)$, $\phi_r = cr(b) \wedge cr(c) \wedge cr(d)$, as described, we get the following three magic rules: $m\_b \leftarrow m\_a$; $m\_c \leftarrow m\_a \wedge cr(b)$ and $m\_d \leftarrow m\_a \wedge cr(b) \wedge cr(c)$.

For a disjunction, the magic rules are even simpler. For every $cr(x)$ in the disjunction, we simply post the magic rule $m\_x \leftarrow m\_a$. Since not all variables in the original rule appear in the condition, some might get removed in the simplification or not be included in the original condition at all. We can ignore them for grounding, but we are interested in their values as soon as we know that the rule can be grounded. Therefore, as soon as the magic variable is created, and ϕ_r is satisfied, we are interested in all the variables in the rule that do not appear in ϕ_r. Finally, we define the modification step for a rule $r = (y \geq f(\bar{x}), y)$, written modify($r$), as changing it to $r = (y \geq f(\bar{x}) \leftarrow m_y, y)$.

magic(r)
  $a := head(r)$
  if($\phi_r = \bigvee_{i=1}^{n} cr(x_i)$)
    for($i \in 1 \ldots n$)
      $P \cup= gen(r) : m\_x_i \leftarrow m\_a$
  if($\phi_r(Q) = \bigwedge_{i=1}^{n} cr(x_i)$)
    $b := m\_a$
    for($i \in 1 \ldots n$)
      $P \cup= gen(r) : m\_x_i \leftarrow b$
      $b := b \wedge cr(x_i)$
  for($v \in vars(r) \setminus (vars(\phi_r) \cup \{a\})$)
    $P \cup= gen(r) : m\_v \leftarrow m\_a \wedge b$
  $P \cup=$ {modify(r)}

The pseudo-code for generation of magic rules and modification of the original rule is given as the function magic that takes a rule as its input. It adds magic rules for a rule to a set P. The first two if conditions handle the disjunctive and conjunctive case respectively. The for loop that follows generates magic rules for variables that are not in ϕ_r. With this function, the entire bottom-up calculation with magic sets is as follows:

1. Create magic variables for all the variables in the program. Call magic for every rule in the program. If the magic rules generated and/or the original rule after modification are not primitive expressions, flatten them.
2. Call ground on the resulting program. While grounding the constraints, build the query by including m_v to Q for every ground variable v that is in some ground constraint.
3. Filter all the magic variables from Q, and magic rules from R. Q and R contain the final set of variables and rules respectively.

The next example demonstrates the complete bottom-up calculation with magic sets.

Example 8. *Consider a BFASP with the following rules:*

$$\begin{array}{ll} R1 & \forall i \in [2,30] \text{ where } i \bmod 2 = 0 : \\ & a[i] \geq b[i-1] + y[i] \\ R2 & \forall i \in [2,30] \text{ where } i \bmod 2 = 0 : \\ & y[i] \geq max(c[2i], d[i+1]) \\ R3 & \forall i \in [1,10] : c[i] \geq 10 \leftarrow s_1[i] \\ R4 & \forall i \in [1,10] : b[i] \geq s_2[i+1] \end{array}$$

where a, b, c, d are arrays of founded integers with ujb of $-\infty$, s_2 is an array of standard Booleans and s_1 is an array of standard integers with domains $(-\infty, \infty)$, and the index set of all arrays is equal to $[1, 100]$. Let us compute ϕ_r for each rule. $\phi_{R_1} = cr(b[i-1]) \wedge cr(y[i])$ $\phi_{R_2} = cr(c[2i]) \vee cr(d[i+1])$, and $\phi_(R_3) = \phi_(R_4) = true$.

We get the following magic rules for these rules:

$$\begin{array}{ll} M1 & gen(R1) : m\_b[i-1] \leftarrow m\_a[i] \\ M2 & gen(R1) : m\_y[i] \leftarrow m\_a[i] \wedge cr(b[i-1]) \\ M3 & gen(R2) : m\_c[2i] \leftarrow m\_y[i] \\ M4 & gen(R2) : m\_d[i+1] \leftarrow m\_y[i] \\ M5 & gen(R3) : m\_s_1[i] \leftarrow m\_c[i] \\ M6 & gen(R4) : m\_s_2[i+1] \leftarrow m\_b[i] \end{array}$$

Let us say we are given the constraint: $a[2] + a[5] \geq 10$. Processing this, we initialize Q with the set $\{m\_a[2], m\_a[5]\}$. Running ground *procedure extends Q with the following variables, the rule used to derived a variable is given in brackets: $m\_b[1](M1)$, $m\_s_2[2](M6)$, $b[1](R4)$, $m\_y[2]$, $m\_c[4](M3)$, $m\_d[3](M4)$, $c[4](R3)$, $m\_s_1[4](M5)$, $y[2](R2)$, $a[2](R1)$. Filtering magic rules, the following ground rules are generated during the grounding (the initial bounds of variables that are not created are plugged in as constants in rules where they appear):*

$$\begin{array}{l} a[2] \leftarrow b[1] + y[2] \\ y[2] \geq max(c[4], -\infty) \\ c[4] \geq 10 \leftarrow s_1[4] \\ b[1] \geq s_2[2] \end{array}$$

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

Unstratified BFASPs

If a given BFASP program is unstratified, then the algorithm described above is not sound. There might be parts of the program that are unreachable from the founded atoms appearing in constraints but are inconsistent. A simple example is a program with a rule $a \leftarrow \neg a$ and a constraint $\neg b$. There are no stable models of this program particularly due to the first rule which can never be satisfied, but the magic set grounding algorithm will ignore the rule since it is not reachable from the b, and therefore wrongly declare $\neg b$ as a stable model of the program. This is *unconditional inconsistency* (Faber, Greco, and Leone 2007). On the other hand, a simple example of *conditional inconsistency* is the following program with three rules: $p \leftarrow \neg q$; $q \leftarrow \neg p$; $y \leftarrow \neg y, \neg q$ and a constraint $\neg p \vee \neg q$. Since we are only interested in p and q, our algorithm will ignore the third rule which means that both $p = true, q = false$ and $q = true, p = false$ are stable assignments, but that is clearly wrong. The second assignment is not stable due to the third rule that we ignored. This is conditional dependency which means that some but not all stable models of the restricted ground program are actual stable models of the complete ground program. The source of both types of inconsistencies is unstratified negation as we can prove that without it, we can safely ignore parts of the program that are not reachable from the constraints. We use a simple strategy to overcome this issue by including all ground magic variables of all array variables that are part of a component in which there is some negative edge between any two of its nodes.

We introduce some notation for the following result that establishes correctness of our approach. Let M be a ground BFASP produced by running the magic set transformation after including the unstratified parts of the program in the initial query for a given non-ground BFASP P. Let G be equal to $grnd(P)$, and let P_i be part of $grnd(P)$ that is not included in M.

Theorem 2. *The stable models of G restricted to the variables $vars(M)$ is equivalent to the stable models of M. That is:*

- *If θ' is a stable model of G, then $\theta'|_{vars(M)}$ is a stable model of M.*
- *If θ is a stable model of M, then there exists θ' s.t. θ' is a stable model of G and $\theta'|_{vars(M)} = \theta$.*

Experiments

We show the benefits of bottom-up grounding and magic sets for computing with BFASPs on a number of benchmarks: *RoadCon*, *UtilPol* and *CompanyCon*.[4] Utilitarian policies *UtilPol* is a problem in which a government decides a set of policies to enact and enacting every policy has a cost. Additionally, there are different citizens and every citizen's happiness depends on two factors: which policies are enacted and how happy some other citizens are. The goal of the problem is to minimize the total cost on enacting policies such that a certain target citizen t is happy above a certain goal. Company controls *CompanyCon* is a problem related to stock markets. The parameters of the problem are the number of companies, each company's ownership of stocks in other companies, and a source company that wants to *control* a destination company. The decision variables are the number of stocks that the source company buys in every other company. A company c controls a company d if the number of stocks that c owns in d plus the number of stocks

[4] All problem encodings and instances can be found at: `www.cs.mu.oz.au/~pjs/bound_founded/`

		Exhaustive		Bottom-up		Magic	
N	SCCs	Flat	Solve	Flat	Solve	Flat	Solve
100	5	4.64	2.12	1.52	0.37	0.32	0.05
300	15	37.42	—	4.05	3.67	0.43	0.16
600	20	240.61	—	20.11	19.81	0.88	0.93
900	30	—	—	30.58	38.86	1.24	2.29
1400	45	—	—	60.61	—	1.87	24.98
1400	20	—	—	266.85	—	3.99	—

Table 2: Road Construction *RoadCon*

Instance				Bottom-up		Magic	
C	P	C_r	P_r	Flat	Solve	Flat	Solve
50	100	5	5	2.13	2.60	0.64	0.02
100	300	10	30	18.40	64.90	3.15	0.09
100	500	10	30	25.82	—	4.96	0.14
250	350	105	105	87.36	—	36.54	21.34
250	400	110	110	91.23	—	40.62	228.03
300	400	125	150	145.02	—	59.56	—

Table 3: Utilitarian Policies *UtilPol*

Instance		Bottom-up		Magic	
C	C_r	Flat	Solve	Flat	Solve
1000	15	23.97	19.48	0.58	1.91
1500	25	55.07	30.05	0.78	4.27
2000	35	93.62	38.74	0.98	7.80
3000	50	215.14	77.40	4.87	21.03
3000	60	212.80	—	5.18	64.14
5000	80	—	—	10.13	88.51

Table 4: Company Controls *CompanyCon*

that other companies that c controls own in d is greater than 50 percent of total number of stocks of company d. The objective of the problem is to minimize the total cost of stocks bought by the source company. All experiments were performed on a machine running Ubuntu 12.04.1 LTS with 8 GB of physical memory and Intel(R) Core(TM) i7-2600 3.4 GHz processor. Our implementation extends MiniZinc 2.0 (LIBMZN) and uses the solver CHUFFED extended with founded variables and rules as described in our previous work (Aziz, Chu, and Stuckey 2013).

Table 2 shows the results for *RoadCon*. N is the number of nodes, and SCCs is the minimum number of strongly connected components in the graph. The edge probability between any two nodes in an SCC is 0.2. We compare exhaustive grounding (simply creating $grnd(P)$) against bottom-up grounding, and bottom-up grounding with magic set transformation. A — represents either the flattener/solver did not finish in 10 minutes or that it ran out of memory. Using bottom-up grounding, the founded variables representing the shortest path between two variables are never created for any variables that are not in the same SCC. Moreover, many useless rules for such variables are also not created. Clearly bottom-up grounding is far superior to naively grounding everything, and magic sets substantially improves on this.

Table 3 shows the results for *UtilPol*. The running time for exhaustive and bottom-up for this benchmark are similar, therefore, the comparison is only given for bottom-up vs. magic sets. In the table, C is the number of citizens, P is the number of policies, C_r represents the maximum number of relevant citizens on which the happiness of t directly or indirectly depends. Similarly, P_r is the maximum number of policies on which the happiness of t and other citizens in C_r depends. This is the part of the instance that is actually relevant to the query, the rest is ignored when magic sets are enabled. It can be seen that magic sets outperform regular bottom-up grounding, especially when the relevant part of the instance is small compared to the entire instance. Note that when P_r is small, the flattening time for magic sets is greater that the solving time since the resulting set of rules is actually simple. This changes, however, as P_r is increased.

Table 4 gives the results for *CompanyCon*. In the table C is the number of total companies while C_r is the maximum number of companies reachable from the destination through the initial ownership graph. The table shows that if C_r is small compared to C, magic sets can give significant advantages. The search strategy used for the problem tries to purchase the least stock possible in each company first, and only increases this when the results don't end up controlling the destination company. Therefore, the complexity of solving the problem is driven by the number of relevant companies, since setting the stock brought for the rest of the companies to zero does not affect the objective. Hence magic sets are less important than in the previous problems. However, as the table shows, the unnecessary founded variables rules can make an instance much more difficult to solve. This is especially true for the flattening time of the bottom-up approach which is severely affected by the increase in the number of irrelevant companies. The same effect is much milder for magic sets since it restricts to only the relevant part of the problem instance.

Conclusion

Bound Founded Answer Set Programming extends Answer Set Programming to disallow circular reasoning over numeric entities. While the semantics of BFASP are a simple generalization of the semantics of ASP, to be practically useful we must be able to model non-ground BFASPs in a high level way. In this paper we show how we can flatten and ground a non-ground BFASP while preserving its semantics, thus creating an executable specification of the BFASP problem. We show that using bottom-up grounding and magic sets transformation we can significantly improve the efficiency of computing BFASPs.

References

Aziz, R. A.; Chu, G.; and Stuckey, P. J. 2013. Stable model semantics for founded bounds. *Theory and Practice of Logic Programming* 13(4–5):517–532. Proceedings of the 29th International Conference on Logic Programming.

Baral, C. 2003. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press.

Faber, W.; Greco, G.; and Leone, N. 2007. Magic sets and their application to data integration. *Journal of Computer and System Sciences* 73(4):584–609.

Frisch, A., and Stuckey, P. 2009. The proper treatment of undefinedness in constraint languages. In Gent, I., ed., *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *LNCS*, 367–382. Springer-Verlag.

Gebser, M.; Schaub, T.; and Thiele, S. 2007. Gringo : A new grounder for answer set programming. In *LPNMR*, 266–271.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference on Logic Programming*, 1070–1080. MIT Press.

Marriott, K., and Stuckey, P. 1998. *Programming with Constraints: an Introduction*. MIT Press.

Mitchell, D. G. 2005. A SAT solver primer. *Bulletin of the EATCS* 85:112–132.

Perri, S.; Scarcello, F.; Catalano, G.; and Leone, N. 2007. Enhancing DLV instantiator by backjumping techniques. *Annals of Mathematics and Artificial Intelligence* 51(2-4):195–228.

Stuckey, P. J., and Tack, G. 2013. Minizinc with functions. In *Proceedings of the 10th International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming*, number 7874 in LNCS, 268–283. Springer.

Syrjnen, T. 2009. *LOGIC PROGRAMS AND CARDINALITY CONSTRAINTS – Theory and Practice*. Ph.D. Dissertation, Faculty of Information and Natural Sciences, Aalto University.