How to Split a Logic Program

Rachel Ben-Eliyahu-Zohary
Department of Software Engineering
JCE- Azrieli College of Engineering
Jerusalem, Israel
rbz@jce.ac.il

Answer Set Programming (ASP) is a successful method for solving a range of real-world applications. Despite the availability of fast ASP solvers, computing answer sets demands a very large computational power, since the problem tackled is in the second level of the polynomial hierarchy. A speed-up in answer set computation may be attained, if the program can be split into two disjoint parts, bottom and top. Thus, the bottom part is evaluated independently of the top part, and the results of the bottom part evaluation are used to simplify the top part. Lifschitz and Turner have introduced the concept of a splitting set, i.e., a set of atoms that defines the splitting.

In this paper, We show that the problem of computing a splitting set with some desirable properties can be reduced to a classic Search Problem and solved in polynomial time. This allows us to conduct experiments on the size of the splitting set in various programs and lead to an interesting discoery of a source of complication in stable model computation. We also show that for Head-Cycle-Free programs, the definition of splitting sets can be adjusted to allow splitting of a broader class of programs.

1 Introduction

Answer Set Programming (ASP) is a successful method for solving a range of real-world applications. Despite the availability of fast ASP solvers, the task of computing answer sets demands extensive computational power, because the problem tackled is in the second level of the polynomial hierarchy. A speed-up in answer set computation may be gained, if the program can be divided into several modules in which each module is computed separately [14, 12, 9]. Lifschitz and Turner propose to split a logic program into two disjoint parts, bottom and top, such that the bottom part is evaluated independently from the top part, and the results of the bottom part evaluation are used to simplify the top part. They have introduced the concept of a splitting set, i.e., a set of atoms that defines the splitting [14]. In addition to inspiring incremental ASP solvers [10], splitting sets are shown to be useful also in investigating answer set semantics [4, 15, 9].

In this paper we raise and answer two questions regarding splitting sets. The first question is, how do we compute a splitting set? We show that if we are looking for a splitting set having a desirable property that can be tested efficiently, we can find it in polynomial time. Examples of desirable splitting sets can be minimum-size splitting sets, splitting sets that include certain atoms, or splitting sets that define a bottom part with minimum number of rules or bottom that are easy to compute, for example, a bottom which is an HCF program [3]. Once we have an effcient algorithm for computing splitting sets, we can use it to investigate the size of a minimal noempty splitting set and discover some interesting results that explain the source of complexity in computing stable models.

Second, we ask if it is possible to relax the definition of splitting sets such that we can now split programs that could not be split using the original definition. We answer affirmatively to the second question as well, and we present a more general and relaxed definition of a splitting set.

2 Preliminaries

2.1 Disjunctive Logic Programs and Stable Models

A propositional *Disjunctive Logic Program* (DLP) is a collection of rules of the form

$$A_1|\dots|A_k \longleftarrow A_{k+1},\dots,A_m, not\, A_{m+1},\dots, not\, A_n, \qquad n \geq m \geq k \geq 0,$$

where the symbol "*not*" denotes negation by default, and each A_i is an atom (or variable). For $k+1 \leq i \leq m$, we will say that A_i appears *positive* in the body of the rule, while for $m+1 \leq i \leq n$, we shall say that A_i appears *negative* in the body of the rule. If $k=0$, then the rule is called *an integrity rule*. If $k>1$, then the rule is called *a disjunctive rule*. The expression to the left of $\longleftarrow$ is called the *head* of the rule, while the expression to the right of $\longleftarrow$ is called the *body* of the rule. Given a rule r, $head(r)$ denotes the set of atoms in the head of r, and $body(r)$ denotes the set of atoms in the body of r. We shall shall sometimes denote a rule by $H \longleftarrow B_{\text{pos}}, B_{\text{neg}}$, where B_{pos} is the set of positive atoms in the body of the rule ($A_{k+1},...,A_m$), B_{neg} is the set of negated atoms in the body of the rule ($A_{m+1},...,A_n$), and H the set of atoms in its head. Given a program $\mathscr{P}$, Lett($\mathscr{P}$) is the set of all atoms that appear in $\mathscr{P}$. From now, when we refer to a program, it is a DLP.

Stable Models [11] of a program $\mathscr{P}$ are defined as Follows: Let Lett($\mathscr{P}$) denote the set of all atoms occurring in $\mathscr{P}$. Let a *context* be any subset of Lett($\mathscr{P}$). Let $\mathscr{P}$ be a *negation-by-default-free* program. Call a context *S closed under* $\mathscr{P}$ iff for each rule $A_1|\dots|A_k \leftarrow A_{k+1},\dots,A_m$ in $\mathscr{P}$, if $A_{k+1},\dots,A_m \in S$, then for some $i=1,\dots,k$, $A_i \in S$. A Stable Model of $\mathscr{P}$ is any minimal context S, such that S is closed under $\mathscr{P}$. A stable model of a general DLP is defined as follows: Let the *reduct of* $\mathscr{P}$ *w.r.t.* $\mathscr{P}$ *and the context S* be the DLP obtained from $\mathscr{P}$ by deleting (i) each rule that has *not* A in its body for some $A \in S$, and (ii) all subformulae of the form *not* A of the bodies of the remaining rules. Any context S which is a stable model of the reduct of $\mathscr{P}$ w.r.t. $\mathscr{P}$ and the context S is a *stable model* of $\mathscr{P}$.

HCF- *Head Cycle Free* programs [3] are DLPs such that in the associated dependency graph there is no cycle including two atoms occurring in the head of the same rule.

Definition 2.1 *A set of atoms S* satisfies *the body of a rule r if all the atoms that appear positive in the body of r are in S and all the atoms that appear negative in r are not in S. A set of atoms S* satisfies *a rule if it does not* satisfy *the body of the rule r or if one of the atoms in head(r) is in S.*

According to [3], a proof of a atom is a sequence of rules that can be used to derive the atom from the program.

Definition 2.2 ([3]) *An atom L has a* proof *w.r.t. a set of atoms S and a logic program* $\mathscr{P}$ *if and only if there is a sequence of rules* $r_1,...,r_n$ *from* $\mathscr{P}$ *such that:*

1. *for all* $1 \leq i \leq n$ *there is one and only one atom in the head of* r_i *that belongs to S.*
2. *L is the head of* r_n*.*
3. *for all* $1 \leq i \leq n$*, the body of* r_i *is satisfied by S.*
4. r_1 *has no atoms that appear positive in its body, and for each* $1 < i \leq n$*, each atom that appears positive in the body of* r_i *is in the head of some* r_j *for some* $1 \leq j < i$*.*

Note that given a proof $r_1,...,r_n$ of some atom L w.r.t. a set of atoms S and a logic program $\mathscr{P}$, for every $1 \leq i \leq n$ $r_1,...,r_i$ is also a proof of some atom L' w.r.t. S and $\mathscr{P}$.

Theorem 2.3 ([3]) *Let* $\mathscr{P}$ *be an HCF DLP. Then a set of atoms S is an answer set of* $\mathscr{P}$ *if and only if S satisfies each rule in* $\mathscr{P}$ *and each* $a \in S$ *has a proof with respect to* $\mathscr{P}$ *and S.*

2.2 Programs and graphs

With every program $\mathscr{P}$ we associate a directed graph, called the *dependency graph* of $\mathscr{P}$, in which (a) each atom in Lett($\mathscr{P}$) is a node, and (b) there is an arc directed from a node A to a node B if there is a rule r in $\mathscr{P}$ such that $A \in body(r)$ and $B \in head(r)$.

A *super-dependency graph SG* is an acyclic graph built from a dependency graph G as follows: For each strongly connected component (SCC) c in G there is a node in SG, and for each arc in G from a node in a strongly connected component c_1 to a node in a strongly connected component c_2 (where $c_1 \neq c_2$) there is an arc in SG from the node associated with c_1 to the node associated with c_2. A program $\mathscr{P}$ is Head-Cycle-Free (HCF), if there are no two atoms in the head of some rule in $\mathscr{P}$ that belong to the same component in the super-dependency graph of $\mathscr{P}$ [3]. Let G be a directed graph and SG be a super dependency graph of G. A *source* in G (or SG) is a node with no incoming edges. By abuse of terminology, we shall sometimes use the term "source" or "SCC" as the set of nodes in a certain source or a certain SCC in SG, respectively, and when there is no possibility of confusion we shall use the term rule for the set of all atoms that appears in the rule. Given a node v in G, scc(v) denotes the set of all nodes in the SCC in SG to which v belongs, and tree(v) denotes the set of all nodes that belongs to any SCC S such that there is a path in SG from S to scc(v). Similarly, when S is a set of nodes, tree(S) is the union of all tree(v) for every $v \in S$. Given a node v in G, scc(v) will be sometimes called the *root* of tree(v). For example, given the super dependency graph in Figure 1, scc(e) $= \{e,h\}$, tree(e) $= \{a,b,e,h\}$, tree($\{f,g\}$) $= \{a,b,c,d,f,g\}$ and tree(r), where $r = c|f \longleftarrow not\, d$ is actually tree($\{c,d,f\}$) which is $\{a,b,c,d,f\}$.

A source in a program will serve as a shorthand for "a source in the super dependency graph of the program." Given a source S of a program $\mathscr{P}$, $\mathscr{P}_{\mathscr{S}}$ denotes the set of rules in $\mathscr{P}$ that uses only atoms from S.

Example 2.4 (Running Example) *Suppose we are given the following program $\mathscr{P}$: $\{$ 1. $a \longleftarrow not\, b$, 2. $e|b \longleftarrow not\, a$, 3. $f \longleftarrow not\, b$, 4. $g|d \longleftarrow c$, 5. $c|f \longleftarrow not\, d$, 6. $h \longleftarrow e$, 7. $e \longleftarrow a, not\, h$, 8. $h \longleftarrow a$ $\}$ In Figure 1 the dependency graph of $\mathscr{P}$ is illustrated in* **solid** *lines. The SG is marked with* **dotted** *lines. Note that $\{a,b\}$ is a source in the SG of $\mathscr{P}$, but it is not a splitting set.*

2.3 Splitting Sets

The definitions of *Splitting Set* and the *Splitting Set Theorem* are adopted from a paper by Lifschitz and Turner [14]. We restate them here using the notation and the limited form of programs discussed in our work.

Definition 2.5 (Splitting Set) *A* Splitting Set *for a program $\mathscr{P}$ is a set of of atoms U such that for each rule r in $\mathscr{P}$, if one of the atoms in the head of r is in U, then all the atoms in r are in U. We denote by $b_U(\mathscr{P})$ the set of all rules in $\mathscr{P}$ having only atoms from U.*

The empty set is a splitting set for any program. For an example of a nontrivial splitting set, the set $\{a,b,e,h\}$ is a splitting set for the program $\mathscr{P}$ introduced in Example 2.4. The set $b_{\{a,b,e,h\}}(\mathscr{P})$ is $\{r_1,r_2,r_6,r_7,r_8\}$.

For the Splitting set theorem, we need the a procedure called Reduce, which resembles many reasoning methods in knowledge representation, as, for example, unit propagation in DPLL and other constraint satisfaction algorithms [5, 6]. Reduce($\mathscr{P}$,X,Y) returns the program obtained from a given program $\mathscr{P}$ in which all atoms in X are set to true, and all atoms in Y are set to false. Reduce($\mathscr{P}$,X,Y) is shown in Figure Reduce. For example, Reduce($\mathscr{P}$,$\{a,e,h\}$,$\{b\}$), where $\mathscr{P}$ is the program from Example 2.4, is

```
Procedure Reduce($\mathscr{P}$,$X$,$Y$)
  Input: A program $\mathscr{P}$ and two sets of atoms: $X$ and $Y$
  Output: An update of $\mathscr{P}$ assuming all the atoms in $X$ are true and all atoms in $Y$ are false
1 foreach atom $a \in X$ do
2     foreach rule $r$ in $\mathscr{P}$ do
3         If $a$ appears negative in the body of $r$ delete $r$ ;
4         If $a$ is in the head of $r$ delete $r$;
5         Delete each positive appearance of $a$ in the body of $r$;
6 foreach atom $a \in Y$ do
7     foreach rule $r$ in $\mathscr{P}$ do
8         If $a$ appears positive in the body of $r$, delete $r$ ;
9         If $a$ is in the head of $r$, delete $a$ from the head of $r$;
10        Delete each negative appearance of $a$ in the body of $r$;
11 return $\mathscr{P}$;
```

the following program (the numbers of the rules are the same as the corresponding rules of the program in Example 2.4): $\{$ 3. $f \longleftarrow$, 4. $g|d \longleftarrow c$, 5. $c|f \longleftarrow$ *not* d $\}$

Theorem 2.6 (Splitting Set Theorem) *(adopted from [14]) Let $\mathscr{P}$ be a program, and let U be a splitting set for $\mathscr{P}$. A set of atoms S is a stable model for $\mathscr{P}$ if and only if $S = X \cup Y$, where X is a stable model of $b_U(\mathscr{P})$, and Y is a stable of Reduce$(\mathscr{P}, X, U - X)$.*

As seen in Example 2.4, a source is not necessarily a splitting set. A slightly different definition of a dependency graph is possible. The nodes are the same as in our definition, but in addition to the edges that we already have, we add a directed arc from a variable A to a variable B whenever A and B are in the head of the same rule. It is clear that a source in this variation of dependency graph must be a splitting set. The problem is that the size of a dependency graph built using this new definition may be exponential in the size of the head of the rules, while we are looking for a polynomial-time algorithm for computing a nontrivial splitting set.

2.4 Search Problems

The area of *search* is one of the most studied and most known areas in AI (see, for example, [16]). In this paper we show how the problem of computing a nontrivial minimum-size splitting set can be expressed as a search problem. We first recall basic definitions in the area of *search*. A *search problem* is defined by five elements: set of states, initial state, actions or successor function, goal test, and path cost. A *solution* is a sequence of actions leading from the initial state to a goal state. Figure 2 provides a basic search algorithm [17].

There are many different strategies to employ when we choose the next leaf node to expand. In this paper we use *uniform cost*, according to which we expand the leaf node with the lowest path cost.

3 Between Splitting Sets and Dependency Graphs

In this section we show that a splitting set is actually a tree in the SG of the program $\mathscr{P}$. The first lemma states that if an atom Q is in some splitting set, all the atoms in scc(Q) must be in that splitting set as well.

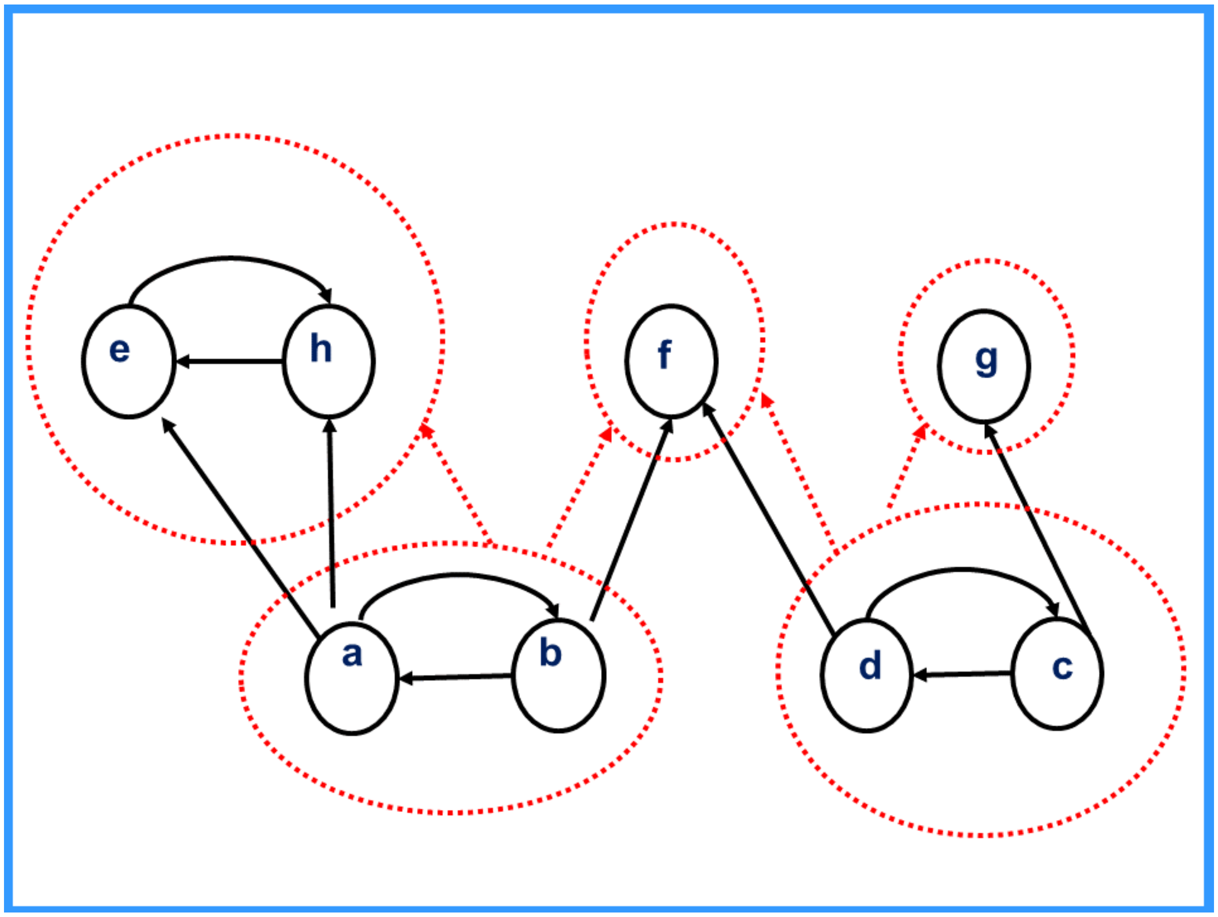
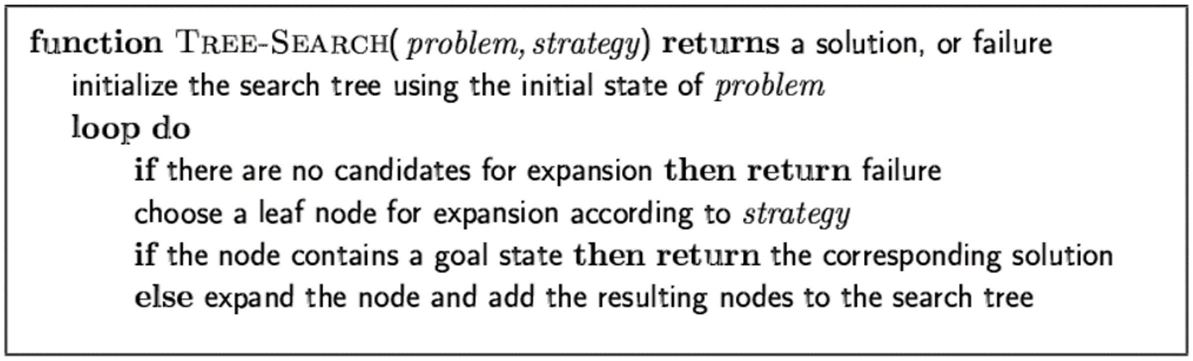


Figure 1: The [super]dependency graph of the program $\mathscr{P}$.



```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
   initialize the search tree using the initial state of problem
   loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

Figure 2: Tree Search Algorithm

Lemma 3.1 *Let $\mathscr{P}$ be a program, let SP be a Splitting Set in $\mathscr{P}$, let $Q \in SP$, and let $S = scc(Q)$. It must be the case that $S \subseteq SP$.*

Proof: Let $R \in S$. We will show that $R \in SP$. Since $Q \in S$, and S is a strongly connected component, it must be that for each $Q' \in S$ there is a path in SG -the super dependency graph of $\mathscr{P}$ - from Q' to Q, such that all the atoms along the path belong to S. The proof goes by induction on i, the number of edges in the shortest path from Q' to Q.

Case $i = 0$. Then $Q = Q'$, and so obviously $Q' \in SP$.

Induction Step. Suppose that for all atoms $Q' \in S$, such that the shortest path from Q' to Q is of size i, Q' belongs to SP. Let R be an atom in S, such that the shortest path from R to Q is of size $i+1$. So, there must be an atom R' such that there is an edge in SG from R to R', and the shortest path from R' to Q is of size i. By the induction hypothesis, $R' \in SP$. Since there is an edge from R to R' in SG, it must be that there is a rule r in $\mathscr{P}$, such that $R \in body(r)$ and $R' \in head(r)$. Since $R' \in SP$ and SP is a Splitting Set, it must be the case that $R \in SP$.

Lemma 3.2 *Let $\mathscr{P}$ be a program, let SP be a Splitting Set in $\mathscr{P}$, let r be a rule in $\mathscr{P}$, and S an SCC in SG – the super dependency graph of $\mathscr{P}$. If $head(r) \cap SP \neq \emptyset$, then $tree(r) \subseteq SP$.*

Proof: We will show that for every $Q \in r$, tree(Q) $\subseteq SP$. Let $Q \in r$. The set tree(Q) is a union of SCCs. We shall show that for every SCC S such that $S \subseteq$ tree(Q), $S \subseteq SP$. Let S' be the root of tree(Q). The proof is by induction on the distance i from S to S'.

Case $i = 0$. Then $S = S'$, and S is the root of tree(Q). Since $head(r) \cap SP \neq \emptyset$, $Q \in r$ and SP is a splitting set, $Q \in SP$. So by Lemma 3.1 $S \subseteq SP$.

Induction Step. Suppose that for all SCCs $S \in$ tree(Q) such that the distance from S to S' is of size i $S \subseteq SP$. Let R be an SCC in tree(r), such that the distance from R to S' is of size $i+1$. So, there must be an SCC R', such that there is an edge in tree(r) from R to R', and the distance from R' to S' is of size i. By the induction hypothesis, $R' \subseteq SP$. Since there is an edge from R to R' in tree(Q), it must be the case that there is a rule r in $\mathscr{P}$, such that an atom from R, say P, is in $body(r)$, and an atom from R', say P', is in $head(r)$. By induction hypothesis, $P' \in SP$, and since SP is a Splitting Set, it must be that $P \in SP$. By Lemma 3.1, $R \subseteq SP$.

Corollary 3.3 *Every Splitting set is a collection of trees.*

Note that the converse of Corollary 3.3 does not hold. In our running example, for instance, tree(g) $= \{c,d,g\}$, but $\{c,d,g\}$ is not a splitting set.

4 Computing a minimum-size Splitting Set as a search problem

We shall now confront the problem of computing a splitting set with a desirable property. We shall focus on computing a nontrivial minimum-size splitting set. Given a program $\mathscr{P}$, this is how we view the task of computing a nontrivial minimum-size splitting set as a search problem. We assume that there is an order over the rules in the program.

State Space. The state space is a collection of forests which are subgraphs of the super dependency graph of $\mathscr{P}$.

Initial State. The empty set.

Actions. 1. The initial state can unite with one of the sources in the super dependency graph of $\mathscr{P}$.

2. A state S, other than the initial state, has only one possible action, which is:
   (a) Find the lowest rule r (recall that the rules are ordered) such that $head(r) \cap S \neq \emptyset$ and Lett$(r) \not\subseteq S$;
   (b) Unite S with tree(r).

Transition Model The result of applying an action on a state S is a state S' that is a superset of S as the actions describe.

Goal Test A state S is a goal state, if there is no rule $r \in \mathscr{P}$ such that $head(r) \cap S \neq \emptyset$ and Lett$(r) \not\subseteq S$. (In other words, a goal state is a state that represents a splitting set.);

Path Cost The cost of moving from a state S to a state S' is $|S'| - |S|$, that is the number of atoms added to S when it was transformed to S'. So, the path cost is actually the number of atoms in the final state of the path.

Once the problem is formulated as a search problem, we can use any of the search algorithms developed in the AI community to solve it. We do claim here, however, that the computation of a nontrivial minimum-size splitting set can be done in time that is polynomial in the size of the program. This search problem can be solved, for example, by a search algorithm called Uniform Cost. Algorithm Uniform Cost [17] is a variation of Dijkstra's single-source shortest path algorithm [7, 8]. Algorithm Uniform Cost is optimal, that is, it returns a shortest path to a goal state. Since the search problem is formulated so that the length of the path to a goal state is the size of the splitting set that the goal state represents, Uniform Cost will find a minimum-size splitting set.

The time complexity of this algorithm is $O(b^m)$, where b is the branching factor of the search tree generated, and m is the depth of the optimal solution. It is easy to see that m cannot be larger than the number of rules in the program, because once we use a rule for computing the next state, this rule cannot be used any longer in any sequel state. As for b, the branching factor, except for the initial state, each state can have at most one child; to generate a child we apply the lowest rule that demonstrates that the current state is not a splitting set. In a given a specific state, the time that required to calculate its child is polynomial in the size of the program. Therefore, this search problem can be solved in polynomial time. This claim is summarized in the following proposition.

Proposition 4.1 *A minimum-size nontrivial splitting set can be computed in time polynomial in the size of the program.*

The following example demonstrates how the search algorithm works, assuming that we are looking for the smallest non-empty splitting set, and we are using uniform cost search.

Example. Suppose we are given the program $\mathscr{P}$ of Example 2.4, and we want to apply the search procedure to compute a nontrivial minimum-size splitting set. The search tree is shown in Figure 3. Our initial state is the empty set. By the definition of the search problem, the successors of the empty set are the sources of the super dependency graph of the program, which in this case are $\{a,b\}$ and $\{c,d\}$, both of which with action cost 2. Since both current leaves have the same path cost, we shall choose randomly one of them, say $\{c,d\}$, and check whether it is a goal state, or in other words, a splitting set. It turns out $\{c,d\}$ is not a splitting set, and the lowest rule that proves it is rule No. 4 that requires a splitting set that includes d to have also c and g. So, we make the leaf $\{c,d,g\}$ the son of $\{c,d\}$ with action cost 1 (only one atom, g, was added to $\{c,d\}$). Now we have two leaves in the search tree. The leaf $\{a,b\}$ with path cost 2, that was there before, and the leaf $\{c,d,g\}$, that was just added, with path cost 3. So we go and check whether $\{a,b\}$ is a splitting set and find out that Rule no. 2 is the lowest rule that proves it is

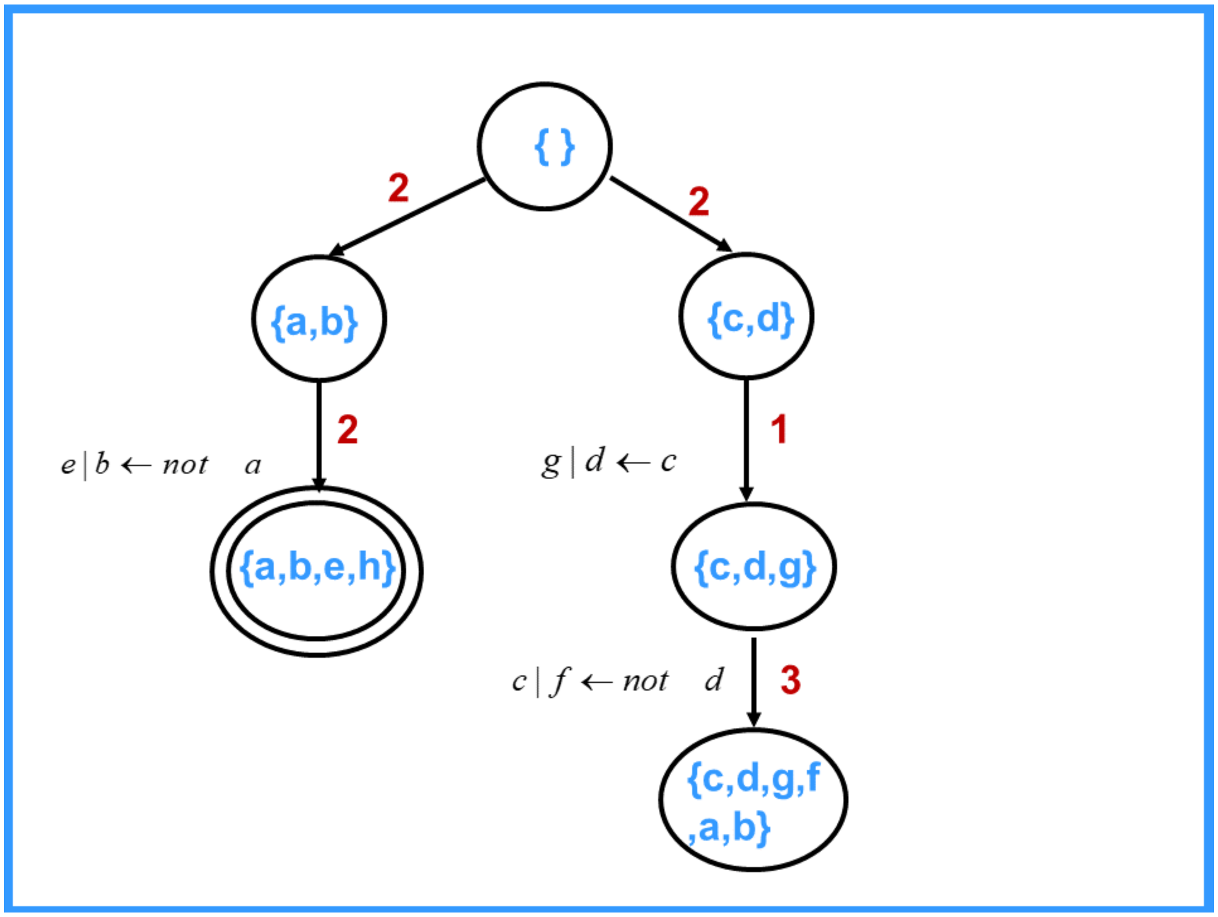


Figure 3: The search tree for $\mathscr{P}$.

not. W,e add the tree of Rule no. 2 and get the child $\{a,b,e,h\}$ with a path cost 4. So, we go now and check whether $\{c,d,g\}$ is a splitting set and find that Rule no. 5 is the lowest rule that proves that it is not. We add the tree of Rule no. 5 and get the child $\{c,d,g,f,a,b\}$ with a path cost 6. Back to the leaf $\{a,b,e,h\}$, the leaf with the shortest path, we find that it is also a splitting set, and we stop the search. $\square$

5 Experiments

We have implemented our algorithm and tested it on randomly generated programs, having no negation as failure. Each rule in the program has exactly 3 variables where any subset of them can be in the head. A stable model is actually a minimal model for this type of program. For each program we have computed a nontrivial minimum-size splitting set. The average nontrivial minimum size of a splitting set, and the median of all nontrivial minimum size splitting sets, as a function of the rules to variable number ratio, are shown in Graph 4 and Graph 5, respectively. The average and median were taken over 100 programs generated randomly, starting with a ratio of 2 and generating 100 random programs for each interval of 0.25. It is clear from the graphs that in the transition value of 4.25 (See [18]) the size of the splitting set is maximal, and it is equal to the number of variables in the program. This is a new way

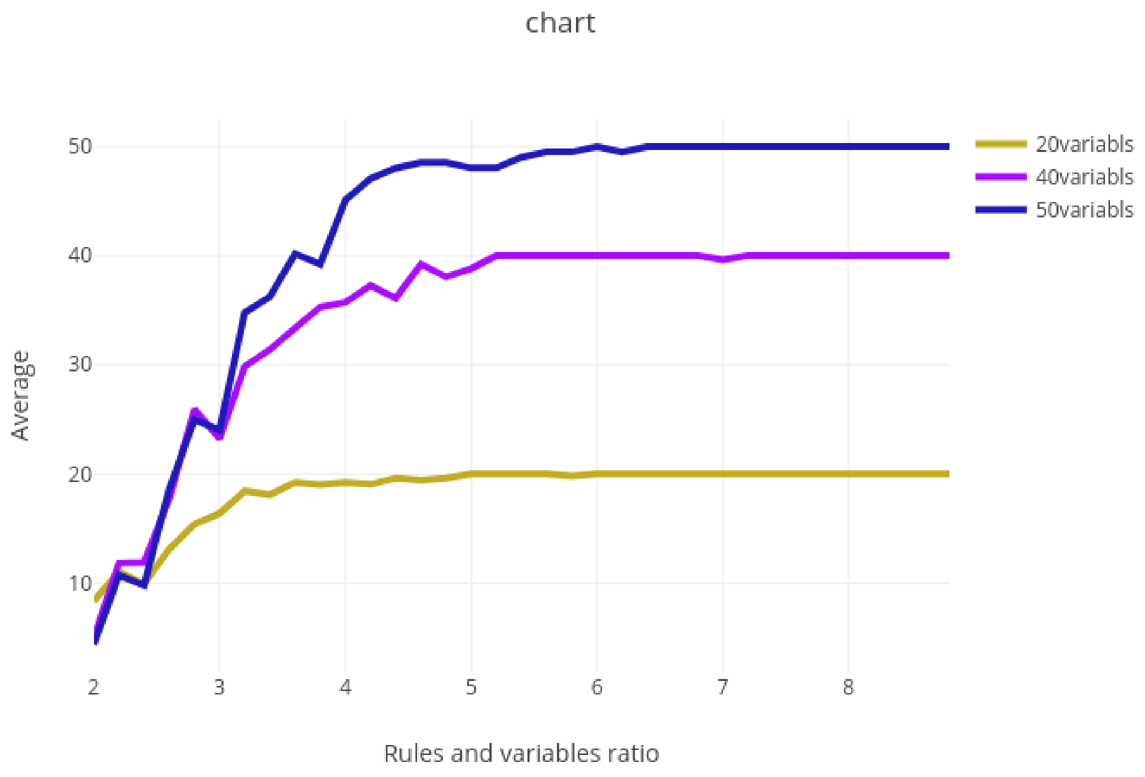


Figure 4: Average size of nonempty splitting sets.

of explaining that, programs in the phase transition value of rules to variable are hard to solve

6 Relaxing the splitting set condition

As the experiments indicate, in the hard random problems the only nonempty splitting set is the set of all atoms in the program. In such cases splitting is not useful at all. In this section we introduce the concept of *generalized splitting set* (g-splitting set), which is a relaxation of the concept of a splitting set. Every splitting set is a g-splitting set, but there are g-splitting sets that are not splitting sets.

Definition 6.1 (Generalized Splitting Set.) *A* Generalized Splitting Set (g-splitting set) *for a program $\mathscr{P}$ is a set of of atoms U such that for each rule r in $\mathscr{P}$, if one of the atoms in the head of r is in U, then all the atoms in the body of r are in U.*

Thus, g-splitting sets that are not splitting sets may be found only when there are disjunctive rules in the program.

Example 6.2 *Suppose we are given the following program $\mathscr{P}$: $\{$ 1. $a \longleftarrow$ not b , 2. $b \longleftarrow$ not a , 3. $b|c \longleftarrow a$, 4. $a|d \longleftarrow b$ $\}$ The program has only the two trivial splitting sets — the empty set and $\{a,b,c,d\}$. However, the set $\{a,b\}$ is a g-splitting set of $\mathscr{P}$.*

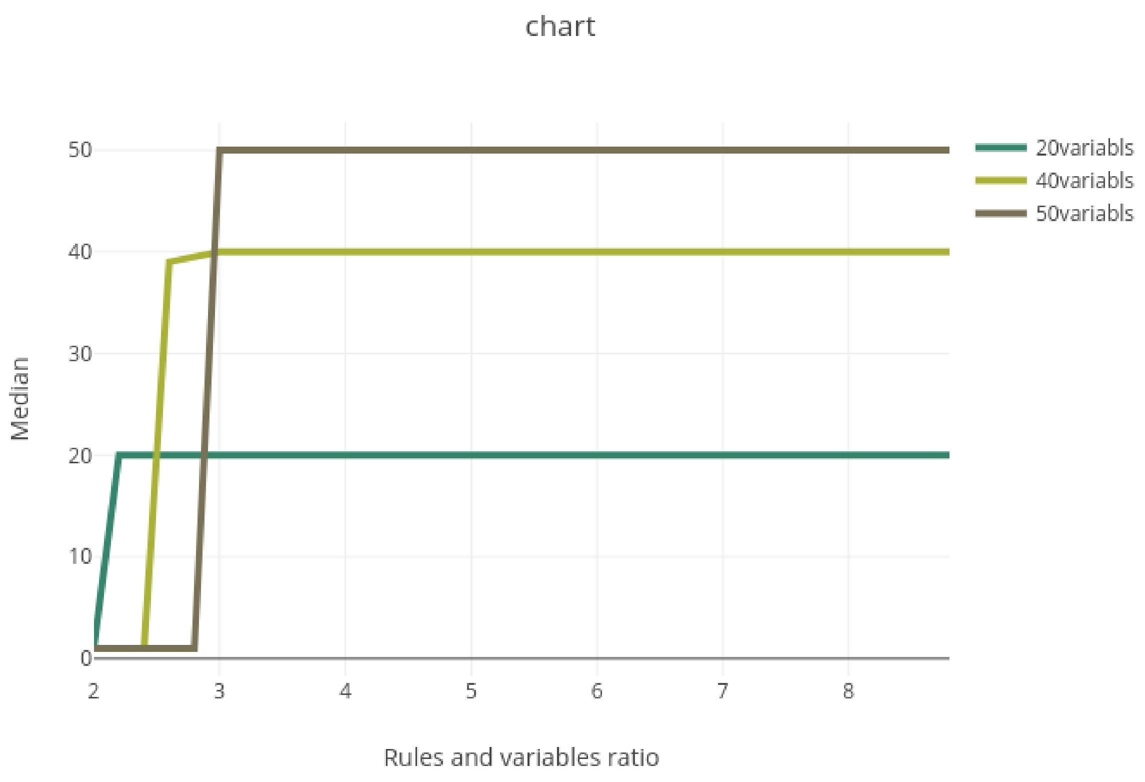


Figure 5: Median size of nonempty splitting sets.

We next demonstrate the usefulness of g-splitting sets. We show that it is possible to compute a stable model of an HCF program $\mathscr{P}$ by computing a stable model of $\mathscr{P}_{\mathscr{S}}$ for a g-splitting set S of $\mathscr{P}$, and then propagating the values assigned to atoms in S to the rest of the program.

Theorem 6.3 (program decomposition.) *Let $\mathscr{P}$ be a HCF program. For any g-splitting-set S in $\mathscr{P}$, let X be a stable model of $\mathscr{P}_S$. Moreover, let $\mathscr{P}' = Reduce(\mathscr{P}, X, S\text{-}X)$, where $Reduce(\mathscr{P}, X, S-X)$ is the result of propagating the assignments of the model X in the program $\mathscr{P}$. Then, for any stable model M' of $\mathscr{P}'$, $M' \cup X$ is a stable model of $\mathscr{P}$.*

Proof: We denote $M' \cup X$ by $M'X$. The proof has two steps. We prove that (1)- $M'X$ is a model of $\mathscr{P}$ and (2) - that every $a \in M'X$ has a proof w.r.t. $\mathscr{P}$ and $M'X$.

Note that it must be the case that $M' \cap X = \emptyset$ and $M' \cap S = \emptyset$.

1. Assume that $M'X$ is not a model of $\mathscr{P}$. Then, there is a rule $r = H \longleftarrow B_{\text{pos}}, B_{\text{neg}}$ in $\mathscr{P}$ such that $M'X$ satisfies the body of r and the head H has empty intersection with $M'X$. Note that r is not in $\mathscr{P}_S$. Otherwise it would not be violated by $M'X$, since X is a model of $\mathscr{P}_S$, no atom in S is in $\mathscr{P}'$ and M' is a stable model of $\mathscr{P}'$.

Since the body of r is satisfied by $M'X$ and $M' \cap X = \emptyset$, B_{pos} can always be written as $(B_{M'} \cup B_X)$, where $B_{M'} = (B \cap M')$, $B_X = (B \cap X)$, and $B_{M'} \cap B_X = \emptyset$, and B_{neg} can always be written as $(B' \cup B_{S-X})$, where $B_{S-X} = (B \cap (S - X))$, $B' = B - B_{S-X}$, and $B' \cap (M' \cup S) = \emptyset$. Analogously, H can be written as $H' \cup H_{S-X}$, where $H_{S-X} = (H \cap (S - X))$, and $H' = H - H_{S-X}$.

After executing procedure `Reduce`$(\mathscr{P}, X, (S - X))$, $\mathscr{P}'$ will contain the rule $r' : H' \longleftarrow B'_{\text{pos}}, B'_{\text{neg}}$, where $B'_{\text{pos}} = B_{M'}$ and $B'_{\text{neg}} = B'$. Since the body of r is satisfied by $M'X$, and $M' \cap S = \emptyset$, it must be the case that the body of r' is satisfied by M'. But, since H has an empty intersection with $M'X$ and $H' \subseteq H$, H' has an empty intersection with $M'X$. So it must be the case that $H' \cap M' = \emptyset$. Thus r' is violated by M', and then M' is not a model of $\mathscr{P}'$ which contradicts the hypothesis. So it must be the case that $M'X$ is a model of $\mathscr{P}$.

2. Next we show that every $a \in M'X$ has a proof w.r.t. $\mathscr{P}$ and $M'X$. First, we show that if $a \in X$, then a has a proof w.r.t. $\mathscr{P}$ and $M'X$. Since $a \in X$ and X is a stable model of $\mathscr{P}_S$, a has a proof w.r.t. $\mathscr{P}_S$ and X. The proof is by induction on the length n of the proof of a w.r.t. $\mathscr{P}_S$ and X.

$n = 1$ So the proof of a is a single rule $r \in \mathscr{P}_S$ of the form $H \longleftarrow B_{\text{neg}}$, where $H \cap X = \{a\}$ and $B_{\text{neg}} \subseteq S - X$. Since $r \in \mathscr{P}_S$ and no atom in S is in $\mathscr{P}'$ it must be the case that $B_{\text{neg}} \cap M'X = \emptyset$ and $H \cap M'X = \{a\}$. In addition, by the definition of $\mathscr{P}_S$, since $r \in \mathscr{P}_S$ $r \in \mathscr{P}$. So r is the proof of a w.r.t. $\mathscr{P}$ and $M'X$.

$n > 1$ We assume that for every $k < n$, if an atom b has a proof of length k w.r.t. $\mathscr{P}_S$ and X, then b has a proof w.r.t. $\mathscr{P}$ and $M'X$. Assume now that for some atom $a \in X$, a has a proof of length n w.r.t. $\mathscr{P}_S$ and X. Let $r \in \mathscr{P}_S$ be the last rule in that proof of a. The rule r must be of the form $H \longleftarrow B_{\text{pos}}, B_{\text{neg}}$, where $H \cap X = \{a\}$, every $b \in B_{\text{pos}}$ has a proof of legth $k < n$ w.r.t. $\mathscr{P}_S$ and X, and $B_{\text{neg}} \subseteq S - X$. By the induction hypothesis, for every $b \in B_{\text{pos}}$ there is a proof of b w.r.t. $\mathscr{P}$ and $M'X$. Since $r \in \mathscr{P}_S$ and no atom in S is in $\mathscr{P}'$ it must be the case that $B_{\text{neg}} \cap M'X = \emptyset$ and $H \cap M'X = \{a\}$. In addition, by the definition of $\mathscr{P}_S$, since $r \in \mathscr{P}_S$ $r \in \mathscr{P}$. So r is the last rule in a proof of a w.r.t. $\mathscr{P}$ and $M'X$.

Second, we show that if $a \in M'$, then a has a proof w.r.t. $\mathscr{P}$ and $M'X$. Since $a \in M'$ and M' is a stable model of $\mathscr{P}'$, a has a proof w.r.t. $\mathscr{P}'$ and M'. The proof is by induction on the length n of the proof of a w.r.t. $\mathscr{P}'$ and M'.

$n = 1$ So the proof of a is a single rule $r \in \mathscr{P}'$ of the form $H \longleftarrow B_{\text{neg}}$, where $H \cap M' = \{a\}$ and $B_{\text{neg}} \cap M' = \emptyset$. Since $r \in \mathscr{P}'$, and by the way *Reduce* works, there must be a rule $r' \in \mathscr{P}$ such that

r' is of the form $H \cup H_{S-X} \longleftarrow B_{\text{pos}}, B'_{\text{neg}}$, where H_{S-X} is the set of atoms in the head of r' that belong to $S - X$, $B_{\text{pos}} \subseteq X$, and $B'_{\text{neg}} = B_{\text{neg}} \cup B_{S-X}$, where B_{S-X} is the set of atoms that appear negative in the body of r' and belong to $S - X$. By Part 1 of this proof, since $B_{\text{pos}} \subseteq X$, every atom in B_{pos} has a proof w.r.t. $\mathscr{P}$ and $M'X$. Considering $B'_{\text{neg}} = B_{\text{neg}} \cup B_{S-X}$, Since $S \cap M' = \emptyset$, $B_{\text{neg}} \cap M'X = \emptyset$, and $B_{S-X} \cap M'X = \emptyset$, so $B'_{\text{neg}} \cap M'X = \emptyset$. In addition, since $H \cap M' = \{a\}$ and $S \cap M' = \emptyset$, $(H \cup H_{S-X}) \cap M' = \{a\}$. So r' is the last rule in a proof of a w.r.t. $\mathscr{P}$ and $M'X$, and hence a has a proof w.r.t. $\mathscr{P}$ and $M'X$.

$n > 1$ We assume that for every $k < n$, if an atom b has a proof of length k w.r.t. $\mathscr{P}'$ and M', then b has a proof w.r.t. $\mathscr{P}$ and $M'X$. Assume now that for some atom $a \in M'$, a has a proof of length n w.r.t. $\mathscr{P}'$ and M'. Let $r \in \mathscr{P}'$ be the last rule in that proof of a. The rule r must be of the form $H \longleftarrow B_{\text{pos}}, B_{\text{neg}}$, where $H \cap M' = \{a\}$, every $b \in B_{\text{pos}}$ has a proof of legth $k < n$ w.r.t. $\mathscr{P}'$ and M', and for each $d \in B_{\text{neg}}$ $d \notin M'$, and by the way *Reduce* works, $d \notin S$. Since $r \in \mathscr{P}'$, and by the way *Reduce* works, there must be a rule $r' \in \mathscr{P}$ such that r' is of the form $H \cup H_{S-X} \longleftarrow B'_{\text{pos}}, B'_{\text{neg}}$, where:

H_{S-X} is the set of atoms in the head of r' that belong to $S - X$,

$B'_{\textbf{pos}} = B_{\textbf{pos}} \cup B_X$, where B_X is a set of atoms that belong to X,

$B'_{\textbf{neg}} = B_{\textbf{neg}} \cup B_{S-X}$, where B_{S-X} is a set of atoms that belong to $S - X$.

We note that:

1. By the induction hypothesis, for every $b \in B_{\text{pos}}$ there is a proof of b w.r.t. $\mathscr{P}$ and $M'X$.
2. By Part 1 of this proof, since $B_X \subseteq X$, every atom in B_{pos} has a proof w.r.t. $\mathscr{P}$ and $M'X$.
3. Since for each $d \in B_{\text{neg}}$ $d \notin M'$, and by the way *Reduce* works, d is also not in S and therefore not in X, d is not in $M'X$.
4. For each $d \in B_{S-X}$ d is not in X and by the way *Reduce* works, d is not in M'. So for every $d \in B_{S-X}$, d is not in $M'X$.
5. Since $H \cap M' = \{a\}$ and no atoms from S is in M', it must be the case that $(H \cup H_{S-X}) \cap M'X = \{a\}$.

From all of the above it follows that r' is the last rule of a proof of a w.r.t. $\mathscr{P}$ and $M'X$.

Consider the program $\mathscr{P}$ from Example 6.2, which has two stable models: $\{a, c\}$ and $\{b, d\}$. Let us compute the stable models of $\mathscr{P}$ according to Theorem 6.3. We take $U = \{a, b\}$, which is a g-splitting set for $\mathscr{P}$. The bottom of $\mathscr{P}$ according to U, denoted $b_{\{a,b\}}(\mathscr{P})$, are Rule 1 and Rule 2, that is: $\{a \longleftarrow \textit{not } b$, $b \longleftarrow \textit{not } a\}$. So the bottom has two stable models: $\{a\}$, and $\{b\}$. If we propagate the model $\{a\}$ to the top of the program, we are left with the rule $\{c \longleftarrow \}$, and we get the stable model $\{a, c\}$. If we propagate the model $\{b\}$ to the top of the program, we are left with the rule $\{d \longleftarrow \}$, and we get the stable model $\{b, d\}$.

7 Related Work

The idea of splitting is discussed in many publications. Here we discuss papers that deal with generating splitting sets and relaxing the definition of a splitting set.

The work in [13] suggests a new way of splitting that introduces a possibly exponential number of new atoms to the program. The authors show that for some typical programs their splitting method is efficient, but clearly it can be quite resource demanding in the worst case.

Baumann [1] discuss splitting sets and graphs, but they do not go all the way in introducing a polynomial algorithm for computing classical splitting sets, as we do here. The authors of [2] suggest *quasi-splitting*, a relaxation of the concept of splitting that requires the introduction of new atoms to the program, and they describe a polynomial algorithm, based on the dependency graph of the program, to efficiently compute a quasi-splitting set. Our algorithm is essentially a search algorithm with fractions of the dependency graph as states in the search space. We do not need the introduction of new atoms to define g-splitting sets.

8 Conclusions

The concept of splitting has a considerable role in logic programming. This paper has two major contributions. First, we show that the task of looking for an appropriate splitting set can be formulated as a classical search problem and computed in time that is polynomial in the size of the program. Search has been studied extensively in AI, and when we formulate a problem as a search problem, we immediately benefit from the library of search algorithms and strategies that has developed in the past and will be generated in the future. Our second contribution is introducing g-splitting sets, which are a generalization of the definition of splitting sets, as presented by Lifschitz and Turner. This allows for a larger set of programs to be split to non-trivial parts.

References

[1] Ringo Baumann (2011): *Splitting an Argumentation Framework.* In James P. Delgrande & Wolfgang Faber, editors: *Logic Programming and Nonmonotonic Reasoning*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 40–53, doi:10.1007/978-3-642-20895-9_6.

[2] Ringo Baumann, Gerhard Brewka, Wolfgang Dvořák & Stefan Woltran (2012): *Parameterized Splitting: A Simple Modification-Based Approach.* In Esra Erdem, Joohyung Lee, Yuliya Lierler & David Pearce, editors: *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 57–71, doi:10.1007/978-3-642-30743-0_5.

[3] Rachel Ben-Eliyahu & Rina Dechter (1994): *Propositional Semantics For Disjunctive Logic Programs.* *Annals of Mathematics and Artificial Intelligence* 12, pp. 53–87, doi:10.1007/BF01530761.

[4] Minh Dao-Tran, Thomas Eiter, Michael Fink & Thomas Krennwallner (2009): *Modular Nonmonotonic Logic Programming Revisited.* In Patricia M. Hill & David S. Warren, editors: *Logic Programming*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 145–159, doi:10.1007/978-3-642-02846-5_16.

[5] Martin Davis, George Logemann & Donald Loveland (1962): *A machine program for theorem-proving.* *Communications of the ACM* 5(7), pp. 394–397, doi:10.1145/368273.368557.

[6] Rina Dechter (2003): *Constraint processing.* Morgan Kaufmann.

[7] Edsger W. Dijkstra (1959): *A note on two problems in connexion with graphs.* *Numerische mathematik* 1(1), pp. 269–271, doi:10.1007/BF01386390.

[8] Ariel Felner (2011): *Position paper: Dijkstra's algorithm versus uniform cost search or a case against dijkstra's algorithm.* In: *Fourth annual symposium on combinatorial search*, pp. 47–51.

[9] Paolo Ferraris, Joohyung Lee, Vladimir Lifschitz & Ravi Palla (2009): *Symmetric splitting in the general theory of stable models.* In: *Twenty-First International Joint Conference on Artificial Intelligence*, pp. 797–803.

[10] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub & Sven Thiele (2008): *Engineering an Incremental ASP Solver.* In Maria Garcia de la Banda & Enrico Pontelli, editors:

Logic Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 190–205, doi:10.1007/978-3-540-89982-2_23.

[11] Michael Gelfond & Vladimir Lifschitz (1991): *Classical Negation in Logic Programs and Disjunctive Databases*. *New Generation Computing* 9, pp. 365–385, doi:10.1007/BF03037169.

[12] Tomi Janhunen, Emilia Oikarinen, Hans Tompits & Stefan Woltran (2009): *Modularity aspects of disjunctive stable models*. *Journal of Artificial Intelligence Research* 35, pp. 813–857, doi:10.1613/jair.2810.

[13] Jianmin Ji, Hai Wan, Ziwei Huo & Zhenfeng Yuan (2015): *Splitting a Logic Program Revisited*. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, AAAI Press, pp. 1511–1517. Available at `http://dl.acm.org/citation.cfm?id=2886521.2886530`.

[14] Vladimir Lifschitz & Hudson Turner (1994): *Splitting a Logic Program*. In: *ICLP*, 94, pp. 23–37.

[15] Emilia Oikarinen & Tomi Janhunen (2008): *Achieving compositionality of the stable model semantics for smodels programs*. *Theory and Practice of Logic Programming* 8(5-6), p. 717–761, doi:10.1017/S147106840800358X.

[16] Judea Pearl (1984): *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Pub. Co., Inc., Reading, MA.

[17] Stuart J. Russell & Peter Norvig (2010): *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education. Available at `http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0136042597,00.html`.

[18] Bart Selman, David G Mitchell & Hector J Levesque (1996): *Generating hard satisfiability problems*. *Artificial intelligence* 81(1-2), pp. 17–29, doi:10.1016/0004-3702(95)00045-3.