TPLP: Page 1–15.

doi:10.1017/S1471068425100331

Splitting a Disjunctive Logic Program

RACHEL BEN-ELIYAHU-ZOHARY
School of Software and Electrical Engineering, Azrieli College of Engineering (JCE), Jerusalem, Israel
(*e-mail:* rbz@jce.ac.il)

submitted 15 August 2025; revised 15 August 2025; accepted 20 August 2025

Abstract

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 significant computational resources, since the problem tackled is on the second level of the polynomial hierarchy. Answer set computation can be accelerated if the program is 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, that is, a set of atoms that defines the splitting.

In a previous paper, the notion of *g-splitting set*, which generalize the concept of splitting sets for disjunctive logic programs, was introduced. In this paper, we further investigate the topic of splitting sets and g-splitting sets. We show that the set inclusion problem for splitting sets can be reduced to a classic Search Problem and solved in polynomial time. We also show that the task of computing g-splitting sets with desirable properties is relatively easy and straightforward. Finally, we show that stable models can be decomposed to models of rules inspired by g-splitting sets and models of the rest of the program. This interesting property can assist in incremental computation of stable models.

KEYWORDS: logic programming, stable model semantics, splitting sets

1 Introduction

Answer Set Programming (ASP) has proven to be an effective approach for addressing a wide variety of real-world problems. Although modern ASP solvers are quite efficient, generating answer sets remains computationally intensive due to the inherent complexity of the problem, which resides at the second level of the polynomial hierarchy. One strategy to accelerate the computation process involves decomposing the logic program into smaller, independently evaluable modules (Lifschitz and Turner (1994); Janhunen *et al.* (2009); Ferraris *et al.* (2009)). In their work, Lifschitz and Turner introduced a method for dividing a program into two non-overlapping sections–referred to as the bottom and top parts–where the bottom can be evaluated independently, and its output is then used to simplify the evaluation of the top. This technique relies on the concept of a splitting set, a collection of atoms that determines how the program is partitioned (Lifschitz and Turner (1994)).

Beyond their utility in enabling incremental ASP solving (Gebser *et al.* (2008)), splitting sets have also contributed to a deeper understanding of answer set semantics (Oikarinen and Janhunen (2008); Dao-Tran *et al.* (2009); Ferraris *et al.* (2009)).

In earlier work (Ben-Eliyahu-Zohary (2021)), the concept of a g-splitting set was introduced, extending the traditional notion of splitting sets to better accommodate disjunctive logic programs. In this paper, we build upon that foundation, offering new insights into both splitting and g-splitting sets. We demonstrate that the set inclusion problem can be transformed into a classical search problem and solved efficiently in polynomial time. Furthermore, we show that identifying g-splitting sets with desirable characteristics is relatively straightforward. Finally, we present a novel result showing that stable models can be divided into models of subsets of rules based on g-splitting sets and models of the remainder of the program – an approach that could facilitate more efficient, incremental computation of stable models for disjunctive logic programs.

The paper is organized as follows. In the next section, we present preliminary definitions and results concerning stable models, program graphs, splitting sets, and search problems in AI. In Section 3, we introduce a polynomial-time algorithm for solving the set inclusion problem. Section 4 analyzes the relationship between g-splitting sets and program graphs. In Section 5, we present the program decomposition lemma for g-splitting sets. Section 6 discusses related work, and Section 7 concludes the paper.

2 Preliminaries

In this section we introduce basic definitions and discuss previous results.

2.1 Disjunctive logic programs and stable models

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

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

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*. We here assume that the program does not contain integrity rules. When the problem of computing stable models is at stake, models can be computed without the integrity rules and then some of them can be eliminated based on the integrity rules. We can also support integrity rules of the form $\longleftarrow X$ via the known simulation $A\longleftarrow \mathit{not}\ A, X$. 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 sometimes denote a rule by $H\longleftarrow B_{\rm pos}, B_{\rm neg}$, where $B_{\rm pos}$ is the set of positive atoms in the body of the rule $(A_{k+1},\ldots,A_m)$, $B_{\rm pos}$ is the set of negated atoms in the body of the rule $(A_{m+1},\ldots,A_n)$, and H the set of atoms in its head. The set of all the atoms that appear in a rule r will be denoted Lett(r). From now on, when we refer to a program, it is a DLP.

https://doi.org/10.1017/S1471068425100331 Published online by Cambridge University Press

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. Stable Models (Gelfond and Lifschitz (1991)) of a program $\mathcal{P}$ are defined as follows: Let Lett($\mathcal{P}$) denote the set of all atoms occurring in $\mathcal{P}$. Let a *context* be any subset of Lett($\mathcal{P}$). Let $\mathcal{P}$ be a *negation-by-default-free* program. Call a context S *closed under* $\mathcal{P}$ if for each rule $A_1 | \ldots | A_k \leftarrow A_{k+1}, \ldots, A_m$ in $\mathcal{P}$, if $A_{k+1}, \ldots, A_m \in S$, then for some $i = 1, \ldots, k$, $A_i \in S$. A Stable Model of $\mathcal{P}$ is any minimal context S, such that S is closed under $\mathcal{P}$. A stable model of a general DLP is defined as follows: Let the *reduct of* $\mathcal{P}$ *w.r.t.* $\mathcal{P}$ *and the context* S be the DLP obtained from $\mathcal{P}$ 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 $\mathcal{P}$ w.r.t. $\mathcal{P}$ and the context S is a *stable model* of $\mathcal{P}$.

Head-Cycle-Free (HCF) programs (Ben-Eliyahu and Dechter (1994)) are DLPs such that in the associated dependency graph there is no cycle including two atoms occurring in the head of the same rule.

2.2 Programs and graphs

With every program $\mathcal{P}$ we associate a directed graph, called the *dependency graph* of $\mathcal{P}$, in which (a) each atom in Lett($\mathcal{P}$) 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 $\mathcal{P}$ 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 SCC c_1 to a node in a SCC 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 $\mathcal{P}$ is Head-Cycle-Free (HCF), if there are no two atoms in the head of some rule in $\mathcal{P}$ that belong to the same component in the super dependency graph of $\mathcal{P}$. HCF programs were first introduced in Ben-Eliyahu and Dechter (1994).

Example 2.1
(Running Example). *Suppose we are given the following program* $\mathcal{P}$:

1. $a \longleftarrow not\ b$
2. $e \mid b \longleftarrow not\ a$
3. $f \longleftarrow not\ b$
4. $g \mid d \longleftarrow c$
5. $c \mid f \longleftarrow not\ d$
6. $h \longleftarrow e$
7. $e \longleftarrow a, not\ h$
8. $h \longleftarrow a$

In Figure 1 the dependency graph of $\mathcal{P}$ *is illustrated in* **solid** *lines. The SG is marked with* **dotted** *lines.*

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

https://doi.org/10.1017/S1471068425100331 Published online by Cambridge University Press

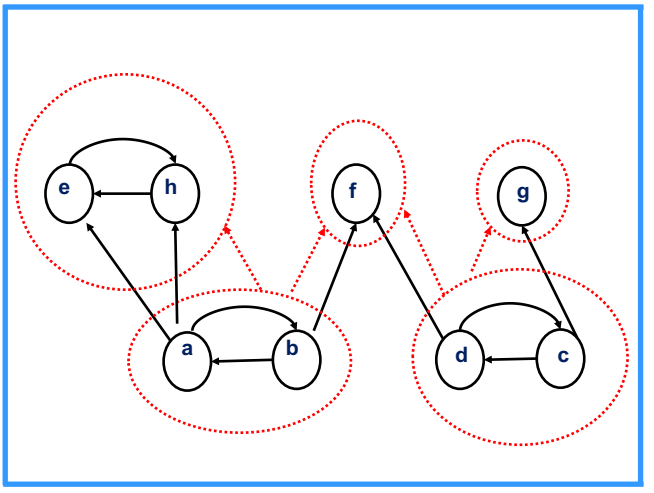

Fig. 1. The [super]dependency graph of the program $\mathcal{P}$.

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, $\mathrm{scc}(v)$ denotes the set of all nodes in the SCC in SG to which v belongs, and $\mathrm{tree}(v)$ denotes the set of all nodes that belongs to any SCC S such that there is a path in SG from S to $\mathrm{scc}(v)$. Similarly, when S is a set of nodes, $\mathrm{tree}(S)$ is the union of all $\mathrm{tree}(v)$ for every $v \in S$. A set of nodes will be called simply a *tree* if it is $\mathrm{tree}(v)$ for some node v. A set of trees will be called *a forest*. Given a node v in G, $\mathrm{scc}(v)$ will be sometimes called the *root* of $\mathrm{tree}(v)$.

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 $\mathcal{P}$, $\mathcal{P}_S$ denotes the set of rules in $\mathcal{P}$ that uses only atoms from S.

Example 2.2
(Running Example continued). *Given the program $\mathcal{P}$ from Example 2.1 above, and the corresponding super dependency graph in Figure 1,* $\mathrm{scc}(e) = \{e, h\}$, $\mathrm{tree}(e) = \{a, b, e, h\}$, $\mathrm{tree}\{f, g\} = \{a, b, c, d, f, g\}$ *and* $\mathrm{tree}(r)$, *where* $r = c|f \longleftarrow$ *not* d *is actually* $\mathrm{tree}\{c, d, f\}$ *which is* $\{a, b, c, d, f\}$. *The set* $\mathcal{P}_{\{a,b,e,h\}}$ *is* $\{r_1, r_2, r_6, r_7, r_8\}$.

2.3 Splitting sets

The definitions of *splitting set* and the *Splitting Set Theorem* are adopted from a paper by Lifschitz and Turner (Lifschitz and Turner (1994)). We restate them here using the notation and the limited form of programs discussed in our work.

Definition 2.3
(splitting set). *A splitting set for a program $\mathcal{P}$ is a set of atoms U such that for each rule r in $\mathcal{P}$, if one of the atoms in the head of r is in U, then all the atoms in r are in 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 $\mathcal{P}$ introduced in Example 2.1.

https://doi.org/10.1017/S1471068425100331 Published online by Cambridge University Press

Algorithm 1 Reduce (P, X, Y)

Input: A program $\mathcal{P}$ and two sets of atoms: X and Y
Output: An update of $\mathcal{P}$ assuming all the atoms in X are true and all atoms in Y are false

```
1  foreach atom a ∈ X do
2      foreach rule r in 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 ∈ Y do
7      foreach rule r in 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 P;
```

For the splitting set theorem, we need the a procedure called Reduce, shown above, which resembles many reasoning methods in knowledge representation, as, for example, unit propagation in DPLL and other constraint satisfaction algorithms (Davis *et al.* (1962); Dechter (2003)). Reduce($\mathcal{P}$, X, Y) returns the program obtained from a given program $\mathcal{P}$ in which all atoms in X are set to true, and all atoms in Y are set to false.

Example 2.4.
Reduce($\mathcal{P}$, $\{a, e, h\}$, $\{b\}$), where $\mathcal{P}$ is the program from Example 2.1, is the following program (the numbers of the rules are the same as the corresponding rules of the program in Example 2.1):

$$\begin{array}{lll} 3. & f & \longleftarrow \\ 4. & g \mid d & \longleftarrow c \\ 5. & c \mid f & \longleftarrow \mathit{not}\ d \end{array}$$

Theorem 2.5
(Splitting Set Theorem). *(adopted from Lifschitz and Turner (1994)) Let $\mathcal{P}$ be a program, and let U be a splitting set for $\mathcal{P}$. A set of atoms S is a stable model for $\mathcal{P}$ if and only if $S = X \cup Y$, where X is a stable model of $\mathcal{P}_U$, and Y is a stable model of* Reduce$(\mathcal{P}, X, U - X)$.

Looking again at Example 2.1, we can see that a source is not necessarily a splitting set. The set $\{a, b\}$ is a source in the SG of $\mathcal{P}$, but it is not a splitting set. For example, see Rule 2 - $e|b \longleftarrow \mathit{not}\ a$. The atom b is in $\{a, b\}$, but the atom e is not in $\{a, b\}$.

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

https://doi.org/10.1017/S1471068425100331 Published online by Cambridge University Press

directed arc from a variable A to a variable B whenever A and B are in the head of the same rule. It is easy to show that a source in this variation of dependency graph must be a splitting set, and every splitting set is a source in this graph. The problem is that the size of a dependency graph built using this new definition is not linear but polynomial in the size of the head of the rules, since each atom in the head of the rule should be connected to every other atom in the same head. So for each head, if there are n atoms in the head, we have $n*(n-1)$ arcs in the graph for this head.

In Ben-Eliyahu-Zohary (2021) it was shown that a splitting set is actually a tree in the SG of the program $\mathcal{P}$.

The following lemma states that if an atom Q is in some splitting set, all the atoms in $\mathrm{scc}(Q)$ must be in that splitting set as well.

Lemma 2.6.
(Ben-Eliyahu-Zohary 2021)
Let $\mathcal{P}$ be a program, let SP be a splitting set in $\mathcal{P}$, let $Q \in SP$, and let $S = \mathrm{scc}(Q)$. It must be the case that $S \subseteq SP$.

The next Lemma states the connection between splitting sets and trees.

Lemma 2.7.
(Ben-Eliyahu-Zohary 2021)
Let $\mathcal{P}$ be a program, let SP be a splitting set in $\mathcal{P}$, let r be a rule in $\mathcal{P}$, and S an SCC in SG – the super dependency graph of $\mathcal{P}$. If $head(r) \cap SP \neq \emptyset$, then $\mathrm{tree}(r) \subseteq SP$.

Corollary 2.8.
(Ben-Eliyahu-Zohary 2021) Every splitting set is a collection of trees.

Note that the converse of Corollary 2.8 does not hold. In our running example, for instance, $\mathrm{tree}(g) = \{c, d, g\}$, but $\{c, d, g\}$ is not a splitting set.

2.4 g-Splitting sets

The concept of g-splitting sets was introduced in Ben-Eliyahu-Zohary (2021).

Definition 2.9
(Generalized Splitting Set). *(Ben-Eliyahu-Zohary 2021) A Generalized Splitting Set (g-splitting set) for a program $\mathcal{P}$ is a set of atoms U such that for each rule r in $\mathcal{P}$, 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.*

Note that in splitting sets, if an atom belongs to the set, then for every rule in which it appears in the head, all atoms appearing in the head of that rule must also be included in the splitting set. In contrast, in g-splitting sets, for every rule in which the atom appears in the head, we require that only the atoms in the *body* of the rule be included in the g-splitting set. Thus, g-splitting sets that are not splitting sets can arise only in programs containing disjunctive rules.

Example 2.10.
Suppose we are given the following program $\mathcal{P}$:

https://doi.org/10.1017/S1471068425100331 Published online by Cambridge University Press

1. $a \longleftarrow not\ b$
2. $b \longleftarrow not\ a$
3. $b \mid c \longleftarrow a$
4. $a \mid d \longleftarrow b$

The program has only the two trivial splitting sets – the empty set and $\{a, b, c, d\}$. *However, the set* $\{a, b\}$, *which is not a splitting set, is a g-splitting set of* $\mathcal{P}$.

In Ben-Eliyahu-Zohary (2021) the following was shown.

Theorem 2.11
(Program decomposition). *Let* $\mathcal{P}$ *be a HCF program. For any g-splitting-set* S *in* $\mathcal{P}$, *let* X *be a stable model of* $\mathcal{P}_S$. *Moreover, let* $\mathcal{P}' =$ Reduce$(\mathcal{P}, X, S - X)$, *where* Reduce$(\mathcal{P}, X, S - X)$ *is the result of propagating the assignments of the model* X *in the program* $\mathcal{P}$. *Then, for any stable model* M' *of* $\mathcal{P}'$, $M' \cup X$ *is a stable model of* $\mathcal{P}$.

In the rest of the paper, when we refer to splitting sets or g-splitting sets we refer to nontrivial sets.

2.5 Search problems

The area of *search* is one of the most studied and most known areas in AI (see, e.g. Pearl (1984)). In a previous paper (Ben-Eliyahu-Zohary (2021)) we have shown how the problem of computing a minimum-size splitting set can be expressed as a search problem. In this paper, we will address the problem of computing a minimum-size splitting set that contains a specific set of atoms. 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. Algorithm 2 is a basic search algorithm (Russell and Norvig (2010)).

Algorithm 2: Tree Search Algorithm

```
Require: problem, strategy
Ensure: solution or failure
  loop
    if there are no candidates for expansion then
      return failure
    else
      choose a leaf node for expansion according to strategy
      if the node contains a goal state then
        return the corresponding solution
      end if
      expand the node and add the resulting nodes to the search tree
    end if
  end loop
```

https://doi.org/10.1017/S1471068425100331 Published online by Cambridge University Press

The algorithm works as follows. It first initializes a tree, starting with a node that represents the initial state. Then it loops forever, unless you either find a solution or exhaust all possibilities. In each step, it first checks whether there are no more nodes to expand. If yes, it returns failure – no solution exists. Otherwise, it uses the chosen strategy to pick the next leaf node (i.e. a node with no children yet). If this node is a goal state, you're done – return the solution! If not, expand the node by generating its successors (next possible states) and adding them to the search tree as children of the node.

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

3 Computing a splitting set having a desirable property

The work of Ben-Eliyahu-Zohary (2021) addresses the problem of computing a splitting set of minimal size. In what follows, we consider the related problem of computing a minimum-size splitting set that also satisfies a desirable property. In this section, we develop an algorithm for computing a splitting set that includes certain atoms. The motivation for this problem is as follows. Consider a situation where the user is interested in the value of certain atoms, that represent the most important pieces of information that she needs at a certain point of time. Then, if we find a splitting set that includes these atoms, we can compute models of only a fraction of the program and hence speed up the computation. We will next call the problem of computing a splitting set that includes a given set of atoms *the set inclusion problem*.

In Ben-Eliyahu-Zohary (2021), the task of computing a minimum-size splitting set was treated as a search problem. Here, we solve the set inclusion problem. This problem is also treated as a search problem and we develop an efficient algorithm for this task as well.

We first define the problem formally.

Definition 3.1
(The set inclusion problem). *The set inclusion problem is defined as follows: given a program $\mathcal{P}$ and a set of atoms B, find a splitting set S for $\mathcal{P}$ such that S includes B and S is minimal - that is - there is no other splitting set S' such that S' includes B and S' is a proper subset of S.*

We now present the algorithm for solving the set inclusion problem. We assume that there is an order over the rules in the program. We denote the set of atoms which we need to include in the splitting set by B.

State Space. The state space is a collection of forests which are subgraphs of the super dependency graph of $\mathcal{P}$ and include tree(B).
Initial State. The initial state is the empty set.
Actions.

1. The initial state unites with the set of all tree(v) such that $v \in B$.
2. A state S, which is not the initial state, has only one possible action, which is computed as follows:

https://doi.org/10.1017/S1471068425100331 Published online by Cambridge University Press

(a) Find the lowest rule r (recall that the rules are ordered) such that $head(r) \cap S \neq \emptyset$ and $\text{Lett}(r) \not\subseteq S$;
(b) Unite S with $\text{tree}(r)$.

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 $r \in \mathcal{P}$ such that $head(r) \cap S \neq \emptyset$ and $\text{Lett}(r) \not\subseteq S$. (In other words, a goal state must 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, which is the size of the splitting set.

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 minimum-size splitting set that includes a certain set of atoms 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 (Russell and Norvig (2010)) is a variation of Dijkstra's single-source shortest path algorithm (Dijkstra (1959); Felner (2011)). 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 the search algorithm is $O(b^m)$, where b is the branching factor of the search tree, and m is the depth of the optimal solution. It is straightforward to see that m cannot exceed the number of rules in the program, since each rule can be used at most once: once a rule is applied to generate a new state, it cannot be reused in any subsequent state. Regarding the branching factor b, note that each state generates at most one child. Specifically, to generate a child state, we apply the lowest-index rule that demonstrates the current state is not a splitting set.

Constructing the dependency graph takes $O(n)$ time, where n is the number of atoms in the program. For a given state S, computing its child requires polynomial time in n. This involves identifying a rule r showing that S is not a splitting set ($O(n^2)$), computing $\text{tree}(r)$ ($O(n)$), and then taking the union of $\text{tree}(r)$ and S ($O(n)$). Therefore, the overall search problem can be solved in polynomial time. This result is formalized in the following proposition.

Proposition 3.2.
The set inclusion problem for splitting sets 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 a minimum-size splitting set that contains the variable g, and we are using uniform cost search.

Example.
Suppose we are given the program $\mathcal{P}$ of Example 2.1, and we want to apply the search procedure to compute a minimum-size splitting set that contains the variable g. Our

https://doi.org/10.1017/S1471068425100331 Published online by Cambridge University Press

initial state is the empty set. By the definition of the search problem, the successor of the empty set is tree(g) in the super dependency graph of the program, which in this case is $\{c, d, g\}$ with action cost 3. Since $\{c, d, g\}$ is the only leaf we now 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. $\{c, d, g, f, a, b\}$ is the only leaf. We find that it is also a splitting set, and we stop the search.

4 Between g-splitting sets and dependency graphs

In Ben-Eliyahu-Zohary (2021) it was shown that every splitting set of a program $\mathcal{P}$ is a forest in the SG of the program $\mathcal{P}$.

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

In this section we show that a *g-splitting set* is also a forest in the SG of the program $\mathcal{P}$. We also show the opposite claim: every forest in SG is a g-splitting set.

The first lemma in this part states that if an atom Q is in some g-splitting set, all the atoms in scc(Q) must be in that g-splitting set as well.

Lemma 4.1.
Let $\mathcal{P}$ be a program, let SP be a g-splitting set in $\mathcal{P}$, let $Q \in SP$, and let $S = \text{scc}(Q)$. 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 SCC, it must be that for each $Q' \in S$ there is a path in SG - the super dependency graph of $\mathcal{P}$ - 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 $\mathcal{P}$, such that $R \in \mathit{body}r$ and $R' \in \mathit{head}(r)$. Since $R' \in SP$ and SP is a g-splitting set, it must be the case that $R \in SP$. □

Lemma 4.2.
Let $\mathcal{P}$ be a program, let SP be a g-splitting set in $\mathcal{P}$, let r be a rule in $\mathcal{P}$, and SG – the super dependency graph of $\mathcal{P}$. If $\mathit{head}(r) \cap SP \neq \emptyset$, then tree($\mathit{body}(r)$) $\subseteq SP$.

Proof.
We will show that for every $Q \in \mathit{body}(r)$, tree(Q) $\subseteq SP$. Let $Q \in \mathit{body}(r)$. 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' in SG.

https://doi.org/10.1017/S1471068425100331 Published online by Cambridge University Press

Case $i = 0$. Then $S = S'$, and S is the root of tree(Q). Since $head(r) \cap SP \neq \emptyset$, $Q \in body(r)$ and SP is a g-splitting set, $Q \in SP$. So by Lemma 4.1 $S \subseteq SP$.

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$(body(r))$, 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$(body(r))$ 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 $\mathcal{P}$, such that an atom from R, say P, is in $bodyr$, and an atom from R', say P', is in $head(r)$. Since $P' \in R'$, and $R' \subseteq SP$, $P' \in SP$, and since SP is a g-splitting set, it must be that $P \in SP$. Since $R =$ scc(P), By Lemma 4.1, $R \subseteq SP$. □

Corollary 4.3.
Every g-splitting set is a forest.

Lemma 4.4.
Let $\mathcal{P}$ be a program. Then S is a g-splitting set for $\mathcal{P}$ if S is a forest in the super dependency graph of $\mathcal{P}$.

Proof.
It was already shown above (See Corollary 4.3) that every g-splitting set is a forest in the super dependency graph of $\mathcal{P}$. It is left to show that every forest in the super dependency graph of $\mathcal{P}$ is a g-splitting set.

Let S be a forest in the super dependency graph of $\mathcal{P}$. We will show that S is a g-splitting set. Assume $v \in S$, and assume there is a rule r in $\mathcal{P}$ such that $v \in head(r)$. Let $v' \in body(r)$. By definition of dependency graph, there is an edge in the graph from v' to v. hence it must be the case that $v' \in S$ as well, since S is a set of trees in the graph. So every forest in the dependency graph of $\mathcal{P}$ must be a g-splitting set.

In the previous section we have presented an algorithm for solving the inclusion problem for splitting sets. What about the set inclusion problem for g-splitting sets? We have shown that every forest is a g-splitting set. We have also shown that if $v \in S$, where v is a variable and S a g-splitting set, then tree(v) must be in S. It follows that given a set of variables A, if we need to compute a minimum-size g-splitting set that contains A we can simply take all tree(v) such that $v \in A$, and this is very easy to compute. □

5 Relaxing the splitting set condition

We now prove that g-splitting sets have the desirable property that splitting sets have. That is, we show that g-splitting sets allow us to split the disjunctive logic programs to several parts and compute each part separately.

Theorem 5.1
(Program decomposition). *Let $\mathcal{P}$ be a program. For any g-splitting-set S in $\mathcal{P}$, let X be a stable model of $\mathcal{P}_S$. Moreover, let $\mathcal{P}'$=Reduce($\mathcal{P}$, X, $S - X$). Then, for any stable model M' of $\mathcal{P}'$, $M' \cup X$ is a stable model of $\mathcal{P}$.*

Proof.
Note that it must be the case that $M' \cap X = \emptyset$. The proof has two steps. We prove that (1) - $(M' \cup X)$ is a model of $\mathcal{P}$ and (2) - that it is minimal.

https://doi.org/10.1017/S1471068425100331 Published online by Cambridge University Press

1. Assume that $(M' \cup X)$ is not a model of $\mathcal{P}$. Then, there is a rule $\delta : H \longleftarrow B$ in $\mathcal{P}$ whose body B is satisfied by $(M' \cup X)$, and the head H has empty intersection with $(M' \cup X)$. Note that δ is not in $\mathcal{P}_S$. Otherwise it would not be violated by $(M' \cup X)$, since X is a model of $\mathcal{P}_S$, and no atom in $\mathcal{P}_S$ is in M'.

Since B is satisfied by $(M' \cup X)$, B can always be written as $(B_{M'} \cup B_X)$, where $B_{M'} = (B \cap M')$, $B_X = (B \cap X)$, and $B_{M'} \cap B_X = \emptyset$. Analogously, since H has empty intersection with $(M' \cup X)$, it can always be written as $H' \cup H_{S-X}$, where $H_{S-X} = (H \cap (S - X))$, and H' is the set of all the other atoms occurring in H.

After executing procedure `Reduce`, $\mathcal{P}'$ will contain the rule $\delta' : H' \longleftarrow B_{M'}$. The set $B_{M'}$ is a subset of M'. But, since H has an empty intersection with $(M' \cup X)$ and $H' \subseteq H$, H' has an empty intersection with M'. Thus δ' is violated by M', and then M' is not a model of $\mathcal{P}'$ which contradicts the hypothesis.

2. Assume that $(M' \cup X)$ is not a minimal model of $\mathcal{P}$. Then there is a non-empty set of atoms A, such that $(M' \cup X) - A$ is a model of $\mathcal{P}$. Let A_X denote the atoms of A belonging to X and $A_{M'}$ the atoms of A belonging to M'. Note that since M' is a minimal model of $\mathcal{P}'$ and $\mathcal{P}'$ has no atoms from X, it must be the case that $M' \cap X = \emptyset$. For A to be non-empty, $A_{M'}$ or A_X has to be non-empty. We prove that in both cases there is a contradiction.

Case $[A_X \neq \emptyset]$: Since X is a minimal model of $\mathcal{P}_S$, $(X - A_X)$ is not a model of $\mathcal{P}_S$. Then, in $\mathcal{P}_S$ there must be a rule $\delta_S : H \longleftarrow B$, such that B is satisfied by $(X - A_X)$ and no atom of H is in $(X - A_X)$. Since δ_S is in $\mathcal{P}_S$, by definition of $\mathcal{P}_S$ no atom of H is outside S, and then no atom of H is in M'. Thus, δ_S is a rule of $\mathcal{P}_S$ (and then of $\mathcal{P}$) which is satisfied by $X - A_X$ (and then by $M' \cup X$) and any atom in the head of δ_S is neither in M' nor in $X - A$. Thus, δ_S is violated by $(M' \cup X) - A$. Since $\delta_S \in \mathcal{P}$, $(M' \cup X) - A$ is not a model of $\mathcal{P}$, a contradiction to the assumption that it is.

Case Case $[A_{M'} \neq \emptyset]$: Since M' is a minimal model of $\mathcal{P}'$, $(M' - A_{M'})$ is not a model of $\mathcal{P}'$. So there must be a rule $\delta' : H \longleftarrow B$ in $\mathcal{P}'$, such that B is satisfied by $(M' - A_{M'})$ and no atom of H is in $(M' - A_{M'})$. Since δ' is in $\mathcal{P}'$, by the way `Reduce` works there must be in $\mathcal{P}$ a rule $\delta : (H \cup H_{S-X}) \longleftarrow (B \cup B_X)$ with B_X a possibly empty subset of X and H_{S-X} a possibly empty subset of $S - X$. Clearly, the body of δ is satisfied by $(M' - A_{M'}) \cup X$, and then, since X and M' are disjoint sets, also by $(M' \cup X) - A)$. However, no atom of δ's head is in $(M' \cup X) - A$. This is because it is assumed that $H \cap M'$ is empty, and $H \cap X$ is empty as well since otherwise, by the way `Reduce` works, δ' would not have been a rule in $\mathcal{P}'$. In addition, $H_{S-X} \cap (M' \cup X) - A)$ is empty. This is because no atoms of $S - X$ are in $\mathcal{P}'$ (and M' is a minimal model of $\mathcal{P}'$), and certainly no atom of X is in $S - X$. Thus, δ is violated by $(M' \cup X) - A$ and hence $(M' \cup X) - A$ is not a model of $\mathcal{P}$, a contradiction to our assumption.

We show that it is possible to compute a stable model of any disjunctive program $\mathcal{P}$, not necessarily HCF program, and then propagating the values assigned to atoms in S to the rest of the program.

https://doi.org/10.1017/S1471068425100331 Published online by Cambridge University Press

The next theorem that we prove claims that stable models of programs can be decomposed to stable models of the part of the program induced by a g-splitting set and a stable model of the rest of the program. This allows for incremental computations of stable models. □

Theorem 5.2
(Stable model decomposition). *Let $\mathcal{P}$ be a program, and let M be a stable model of $\mathcal{P}$. Moreover, assume there is a g-splitting set S in $\mathcal{P}$ such that $X = M \cap S$ is a stable model of $\mathcal{P}_S$, and let $\mathcal{P}' = Reduce(\mathcal{P}, X, S - X)$. Then $M - X$ is a stable model of $\mathcal{P}'$.*

Proof.
We first show that $M' = M - X$ is a model of $\mathcal{P}'$. Let $H \longleftarrow B \in \mathcal{P}'$ and assume M' satisfies B. Since $\mathcal{P}'$ was obtained from $\mathcal{P}$ using the procedure `Reduce`, and by the way `Reduce` works, there must be a possibly empty sets B_X and H_{S-X} such that B_X is satisfied by X and $H_{S-X} \subseteq S - X$, and such that $(H \cup H_{S-X}) \longleftarrow (B \cup B_X) \in \mathcal{P}$, and $H \cap X = \emptyset$. Since M' satisfies B and X satisfies B_X, M satisfies $B \cup B_X$, and since M must satisfy the rule $(H \cup H_{S-X}) \longleftarrow (B \cup B_X)$, In M there is at least one atom from $(H \cup H_{S-X})$. Since $H \cap X = \emptyset$ and $(H_{S-X} \cap X) = \emptyset$, $(H \cup H_{S-X}) \subseteq M - X$. Hence $H \longleftarrow B$ is satisfied by M'.

We now show that $M' = M - X$ is a *minimal* model of $\mathcal{P}'$. Assume conversely that it is not. Then there must be a non-empty subset of atoms $W \subseteq M'$ such that $M' - W$ is a model of $\mathcal{P}'$. Note that since $M' = M - X$ and $X = M \cap S$, $M' \cap S = \emptyset$. So it must be the case that W, which is a subset of M', does not have atoms from S. We show that $M - W$ is model of $\mathcal{P}$, a contradiction to M being a minimal model of $\mathcal{P}$. Let $\delta = H \longleftarrow B \in \mathcal{P}$ and assume $M - W$ satisfies B. We will show that $H \cap (M - W) \neq \emptyset$, and so $M - W$ satisfies δ. The head H of δ may be written as $H' \cup H_S$, where $H_S = H \cap S$, and $H' = H - S$. The body B of δ may be written as $B' \cup B_S$, where $B_S = B \cap S$ and $B' = B - S$. If H' is empty, then δ is of the form $H_S \longleftarrow B$, and hence $\delta \in \mathcal{P}_S$ (remember that S is a g-splitting set). Since M satisfies δ and $W \cap S$ is empty, $M - W$ must satisfy δ as well. If H' is not empty, by the way `Reduce` works, the rule $H' \longleftarrow B'$ must belong to $\mathcal{P}'$. Since $M - W$ satisfies B', $B' = B - S$ and $X \subseteq S$, it must be the case that $M - X - W$ satisfies B'. Since $M - X - W$ is a model of $\mathcal{P}'$, it must be the case that $H' \cap (M - X - W) \neq \emptyset$. So clearly $H' \cap (M - W) \neq \emptyset$. Since $H' \subseteq H$, $H \cap (M - W) \neq \emptyset$, so $M - W$ satisfies δ, a contradiction to M being a minimal model of $\mathcal{P}$. □

6 Related work

The idea of splitting has been discussed in many publications. Here, we focus on papers that address the generation of splitting sets and the relaxation of the definition of a splitting set.

The work in Ben-Eliyahu-Zohary (2021) is highly relevant to this paper and has been referenced throughout. The paper Angiulli *et al.* (2022) presents similar ideas regarding the decomposition of a theory to compute a minimal model. However, Angiulli *et al.* (2022) addresses propositional theories rather than DLPs. Moreover, it does not provide algorithms for computing s-splitting sets or introduce the concept of g-splitting sets.

https://doi.org/10.1017/S1471068425100331 Published online by Cambridge University Press

The work in Ji *et al.* (2015) proposes a novel splitting method that may introduce an exponential number of new atoms into the program. The authors demonstrate that their method is efficient for certain typical programs, but it can be quite resource-intensive in the worst case.

Baumann (2011) discusses splitting sets and graphs, but does not provide a complete method for computing classical splitting sets using a polynomial-time algorithm, as we do here. The authors of Baumann *et al.* (2012) propose *quasi-splitting*, a relaxation of the splitting set concept that requires the introduction of new atoms into the program. They also present a polynomial-time algorithm, based on the program's dependency graph, to efficiently compute a quasi-splitting set.

Our approach, by contrast, is essentially a search algorithm, where states in the search space correspond to fragments of the dependency graph. Unlike quasi-splitting, our method does not require the introduction of new atoms to define g-splitting sets.

7 Conclusions

The concept of splitting plays a significant role in logic programming.

This paper makes several contributions. First, we show that the task of set inclusion for splitting sets can be formulated as a classical search problem and solved in time that is polynomial in the size of the program.

Search has been extensively studied in AI. By formulating the problem in terms of search, we benefit from the extensive library of search algorithms and strategies that have been developed over the years – and will continue to evolve.

Second, we further investigate the notion of g-splitting sets, a generalization of the splitting set definition originally introduced by Lifschitz and Turner.

This generalization enables a broader class of programs to be split into nontrivial parts.

Finally, we show that g-splitting sets possess several useful properties that are easy to compute. In the future, we plan to implement the promising algorithms developed in this work and evaluate their performance empirically. In particular, it will be important to assess whether g-splitting sets offer practical advantages in computing stable models.

References

Angiulli, F., Ben-Eliyahu-Zohary, R., Fassetti, F. and Palopoli, L. 2022. Graph-based construction of minimal models. *Artificial Intelligence* 313, 103754.

Baumann, R. 2011. Splitting an argumentation framework. In *Logic Programming and Nonmonotonic Reasoning*, Delgrande, J. P. and Faber, W. (ed.), Berlin, Heidelberg: Springer, 40–53.

Baumann, R., Brewka, G., Dvořák, W. and Woltran, S. 2012. *Parameterized Splitting: A Simple Modification-Based Approach*. Springer, Berlin, Heidelberg, 57–71.

Ben-Eliyahu, R. and Dechter, R. 1994. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence* 12, 53–87.

Ben-Eliyahu-Zohary, R. 2021. How to split a logic program. In Proceedings 37th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2021, Porto (virtual event), 20–27th September 2021, volume 345 of EPTCS, Formisano, A., Liu, Y. A., Bogaerts, B., Brik, A., Dahl, V., Dodaro, C., Fodor, P., Pozzato, G. L., Vennekens, J. and Zhou, N.-F., Eds. 27–40.

https://doi.org/10.1017/S1471068425100331 Published online by Cambridge University Press

Dao-Tran, M., Eiter, T., Fink, M. and Krennwallner, T. 2009. Modular nonmonotonic logic programming revisited. In *Logic Programming*, Hill, P. M. and Warren, D. S., Eds. Springer, Berlin, Heidelberg, 145–159.

Davis, M., Logemann, G. and Loveland, D. 1962. A machine program for theorem-proving. *Communications of the ACM* 5, 7, 394–397.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1, 1, 269–271.

Felner, A. 2011. Position paper: Dijkstra's algorithm versus uniform cost search or a case against dijkstra's algorithm. In Fourth Annual Symposium on Combinatorial Search.

Ferraris, P., Lee, J., Lifschitz, V. and Palla, R. 2009. Symmetric splitting in the general theory of stable models. In Twenty-First International Joint Conference on Artificial Intelligence.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T. and Thiele, S. 2008. Engineering an incremental asp solver. In *Logic Programming*, de la Banda, M. G. and Pontelli, E., Eds. Springer, Berlin, Heidelberg, 190–205.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing* 9, 365–385.

Janhunen, T., Oikarinen, E., Tompits, H. and Woltran, S. 2009. Modularity aspects of disjunctive stable models. *Journal of Artificial Intelligence Research* 35, 813–857.

Ji, J., Wan, H., Huo, Z. and Yuan, Z. 2015. Splitting a logic program revisited. In Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, AAAI'15. AAAI Press, 1511–1517.

Lifschitz, V. and Turner, H. 1994. Splitting a logic program. *ICLP* 94, 23–37.

Oikarinen, E. and Janhunen, T. 2008. Achieving compositionality of the stable model semantics for smodels programs. *Theory and Practice of Logic Programming* 8, 5-6, 717–761.

Pearl, J. 1984. Heuristics: intelligent search strategies for computer problem solving.

Russell, S. J. and Norvig, P. 2010. *Intelligence – A Modern Approach*. Third International Edition. Pearson Education.

https://doi.org/10.1017/S1471068425100331 Published online by Cambridge University Press