Graph Based Answer Set Programming Solver Systems *

Fang Li, Elmer Salazar, Gopal Gupta
University of Texas at Dallas
Richardson, USA
{fang.li, elmer.salazar, gupta}@utdallas.edu

1 Dependency Graph

A dependency graph [3] uses nodes and directed edges to represent dependency relationships of an ASP rule.

Definition 1. *The dependency graph of a program is defined on its literals s.t. there is a positive (resp. negative) edge from p to q if p appears positively (resp. negatively) in the body of a rule with head q.*

Conventional dependency graphs are not able to represent ASP programs uniquely. This is due to the inability of dependency graphs to distinguish between non-determinism (multiple rules defining a proposition) and conjunctions (multiple conjunctive sub-goals in the body of a rule) in logic programs. For example, the following two programs have identical dependency graphs (Figure 1).

```
%% program 1
p :- q, not r, not p.
```

```
%% program 2
p :- q, not p. p :- not r.
```

To make conjunctive relationships representable by dependency graphs, we first transform it slightly to come up with a novel representation method. This new representation method, called conjunction node representation (CNR) graph, uses an artificial node to represent conjunction of sub-goals in the body of a rule. This conjunctive node has a directed edge that points to the rule head (Figure 2).

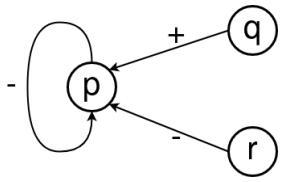
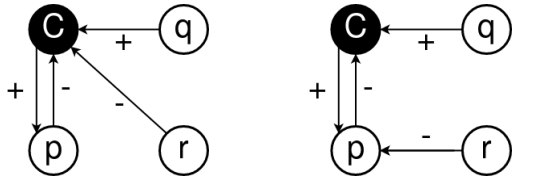


Figure 1: Dep. Graph for Programs 1 & 2



(a) CNR for Program 1 (b) CNR for Program 2

Figure 2: CNRs for Program 1 & 2

The conjunction node, which is colored black, refers to the conjunctive relation between the incoming edges from nodes representing subogals in the body of a rule. Note that a CNR graph is not a conventional dependency graph.

Converting CNR Graph to Dependency Graph Since CNR graph does not follow the dependency graph convention, we need to convert it to a proper dependency graph in order to perform dependency

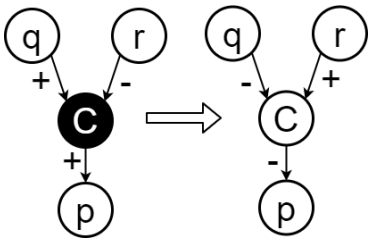
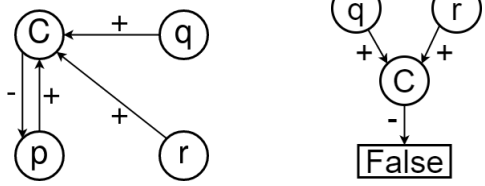


Figure 3: CNR-DG Transformation



(a) Program 3 (b) Program 4

Figure 4: Constraint DG

graph-based reasoning. We use a simple technique to convert a CNR graph to an equivalent conventional dependency graph. We negate all in-edges and out-edges of the conjunction node. This process essentially converts a conjunction into a disjunction. Once we do that we can treat the conjunction node as a normal node in a dependency graph. As an example, Figure 3 shows the CNR graph to dependency graph transformation for program `p :- q, not r.` This transformation is a simple application of De Morgan's law. The rule in this program represents `p :- C. and C :- q, not r.` The transformation produces the equivalent rules `p :- not C., C :- not q. and C :- r.`
Since conjunction nodes are just helper nodes which allow us to perform dependency graph reasoning, we don't report them in the final answer set.

Constraint Representation ASP also allows for special types of rules called constraints. There are two ways to encode constraints: (i) headed constraint where negated head is called directly or indirectly in the body (e.g., Program 3), and (ii) headless constraints (e.g., Program 4).

```
%% program 3
p :- not q, not r, not p.
```

```
%% program 4
:- not q, not r.
```

Our algorithm models these constraint types separately. For the former one, we just need to apply the CNR-DG transformation directly. Note that the head node connects to the conjunction node both with an in-coming edge and an out-going edge (Figure 4a). For the headless constraint, we create a head node with truth value as *False*.

The reason why we don't treat a headless constraint the same way as a headed constraint is because in the latter case, if head node (p in Program 3) is provable through another rule, then the headed constraint is inapplicable. Therefore, we cannot simply assign a false value to its head.

2 grASP: A Bottom-up Approach

We have developed the grASP graph-based algorithm for finding answer sets. The philosophy of grASP is to translate an ASP program into a dependency graph (via CNR conversion), then propagate truth values from nodes whose values are known to other connected nodes, obeying the sign on the edge, until the values of all the nodes are fixed. However, due to possible existence of a large number of cycles, the propagation process is not straightforward. In grASP, we define a collection of rules for propagating values among nodes involved in cycles. These assignment rules take non-monotonicity of answer set program and the causal relationship among nodes in the dependency graph into account.

Unlike other SAT-solver based approaches, our graph based approach enables stratification of ASP programs on the basis of dependence. The Splitting Theorem [2] can thus be used to link the various levels, permitting values to be propagated among nodes more efficiently. Also, the existence of sub-

structures (sub-graphs) makes an efficient recursive implementation algorithm possible.

2.1 The grASP Algorithm

The grASP algorithm is recursive in nature. Since a dependency graph represents the causal relationships among nodes, the reasoning should follow a topological order. We don't need to do topological sorting to obtain the order, instead, for each iteration, we just pick those nodes which have no in-coming edges. We call this kind of node a **root node**. After picking the root nodes, the algorithm checks their values. If a root node's value has not been fixed (no value yet), we assign *False* to it. Otherwise, the root node will keep its value as is. Once all root nodes' values are fixed, we will propagate the values along their out-going edges in accordance with the sign on each edge (the propagation rules will be discussed in Section **??**). At the end of this iteration, we remove all root nodes from the graph, then pass the rest of the graph to the recursive call for the next iteration.

The input graph may contain cycles, and, of course, there will be no root nodes in a cycle. Therefore, this recursive process will leave a cycle unchanged. To cope with this issue, we proposed a novel solution, which wraps all nodes in the same cycles together, and treat the wrapped nodes as a single *virtual* node. All the in-coming and out-going edges connecting the wrapped nodes to other nodes will be incident on or emanate from the virtual node. Thus, the graph is rendered acyclic and ready for the root-finding procedure.

For each iteration of the recursive procedure, we have to treat regular root nodes and virtual root nodes differently. If the node is a regular node, we do the value assignment, but if it is a virtual node, we will have to *break the cycles*. Cycle breaking means that we will remove the appropriate cycle edges by assigning truth values to the nodes involved (*cycle breaking* will be discussed in this section later). After cycle breaking, we will pass the nodes and edges in this virtual node to another recursive call, because the virtual node can be seen as a substructure of the program. The returned value of the recursive call will be the answer set of the program constituting the virtual node. When all regular and virtual root nodes are processed, we will have to merge the values for propagation.

The value propagation in each iteration makes use of the **splitting theorem** [2] (details omitted due to lack of space). After removing root nodes, rest of the graph acts as the *top* strata and all of the predecessors constitute the *bottom* strata, using the terminology of [2]. Thus, when we reach the last node in the topological order, we will get the whole answer set.

The cycle breaking procedure may return multiple results, because a negative even cycle generates two worlds. Therefore, the merging of solution for the root nodes may possibly result in exponential number of solutions. For example, if the root nodes consists of one regular node and two virtual nodes, each virtual node generates two worlds & the merging process will return four worlds. Of course, this exponential behavior is inherent to ASP.

More details about grASP can be found at arXiv [1].

3 igASP: A Top-down Approach

Our top-down approach is called igASP, which stands for incremental graph-based ASP solver. The philosophy of igASP is to translate an ASP program into a CNR dependency graph, which is always constrained by some constraint rules, then try to satisfy the constraints by assigning presumed truth values to the related nodes, until all constraints have been satisfied. At the same time, igASP will propagate truth values of the nodes whose truth value has already been determined.

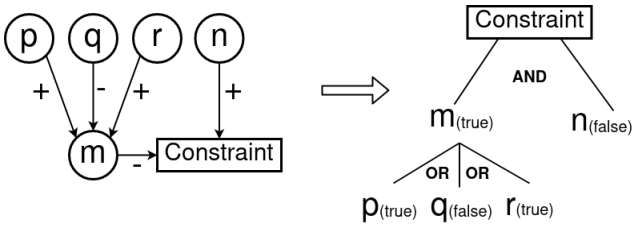


Figure 5: Satisfiability Example

Our graph-based approach performs reasoning in an incremental manner. It starts from the constraints in the answer set program and traces along causal nodes until it find support through facts (well-founded case) or it detects a cycle through negation (cyclic case). Our algorithm can be thought of as a more general form of the Galliwasp algorithm for query-driven execution of answer set programs [4]. The igASP approach is constraint-driven and thus significantly reduces the search space by avoiding exploration of worlds that are inconsistent with the constraints. Furthermore, the incremental reasoning from constraints allows igASP to perform query-driven execution.

3.1 The igASP Algorithm

The igASP algorithm is a recursive algorithm. Since a CNR dependency graph represents the causal relationships among nodes, a topological order would indicate the truth values flow along edges from one node to another starting from the leaves. By their nature, the constraint nodes (labeled *False*, discussed in Section 1) will be at the end of such flows in the CNR dependency graph. Therefore, we can incrementally establish the satisfiability relationships across all the nodes starting from the constraint nodes. This incremental establishment of satisfiability starting from the constraint nodes amounts to developing a proof tree. An example (Program 5) is shown in Figure 5 where the graph is to the left and proof tree to the right. To falsify the constraint node, i.e., to ensure it is *False*, node *m* must be *True* and *n* must be *False*. For node *m* to be *True*, at least one of the three must hold: *p* is *True*, *q* is *False*, or *r* is *True*. When every node's presumed truth value has been found to be consistent with all the dependencies, the algorithm will return the answers.

```
%% program 5
m :- p. m :- not q. m :- r. :- not m. :- n.
```

Effective Edge: An effective edge in a CNR dependency graph refers to any edge that propagates *True* value to the node it is incident on. There are two type of *effective* edges: (i) positive edge emanating from a *True* node; (ii) negative edge emanating from a *False* node. An effective edge only points to a *True* node.

Satisfying Conjunction vs. Disjunction: There are two kinds of dependencies that may arise for a node in the CNR dependency graph: conjunctive and disjunctive. A conjunctive dependency refers to the situation where a node is presumed to be *False*. In this case, none of the edges incident into the node should be effective edges. A disjunctive dependency indicates that when a node is presumed to be *True*, at least one of the in-coming edges should be effective. Since igASP works in a reverse manner (from constraints to facts), we may get multiple partial models before we can validate the *True*/*False* label of the current node. For both conjunction and disjunction, these partial models need to be merged for the sake of integrity as well as efficiency. The merging process is discussed later.

Proof Branch: In the igASP algorithm, we start from the constraints that have to be shown to be false, and incrementally construct a proof tree obeying the constraints imposed by the CNR dependency graph. In this incremental reasoning process, we will pursue various paths in the CNR dependency graph. Our proof will have multiple branches, corresponding to various paths in the CNR dependency graph. Traversal of a branch stops when we reach a fact node whose value has already been given by the ASP program (i.e., known to be a true due to being a fact or known to be false because the atom does not have a rule with that atom as head), or sense that the branch contains a **cycle**.

Cycle Handling: For positive cycles, we need to ensure that any models computed are consistent with ASP semantics. Suppose we have a program `p :- q.  q :- p.`, under ASP semantics, it will have only one answer set: {p/False, q/False}. The other model ({p/True, q/True}) has to be rejected, as it is not well-founded per ASP semantics. Thus, positive cycles have to be handled properly so that only correct answer sets are reported.

To detect a cycle, igASP keeps track of the presumed nodes along the branch, when the current node has been seen previously, we will check whether there exist any *False* node between these two nodes. If so, it is an even cycle, otherwise, it is a positive cycle and only the falsifying assignment should be computed.

Model Merging As mentioned previously, for each presumed node n (i.e., a node assigned a truth value), its dependencies will be either conjunctive (if n is presumed *False*) or disjunctive. (if n is presumed *True*). For both conditions, we need to merge the partial models that have been computed so far while assigning a truth value to the dependent nodes. For the conjunctive condition, the merging process only takes successfully merged models, each of which are the union of two non-conflicting sub-models. For example, consider a node n that is presumed to be *False*. Suppose it has two predecessors p and q, both p and q connect to n via negative edges. So that n will only be *False* when both p and q will be *True*. We need a conjunctive merge here. Suppose we have sub-models {p1:{a/True, d/True, b/False}, p2:{a/False, b/True}} that hold for p to be *True*, and sub-models {q1:{a/True, c/True, b/False}} for q to be *True*. The conjunction merging of sub-models between p and q will only accept the union of $p1$ and $q1$, because $p2$ conflicts with $q1$. Therefore, there will only be one model to satisfy for n being *False*, that is {a/True, c/True, d/True, b/False}.

For a disjunctive merging, we will keep the conflicted sub-models along with successfully merged ones. Let's modify the above example a little bit by presuming the value of node n to be *True*, and keep everything else unchanged. Now the merging condition became disjunctive, because one of p or q being *False* will still make n *True*. Since $p1$ and $q1$ can be merged without conflict, we replace them by their union {a/True, c/True, d/True, b/False}. But this time we don't discard $p2$, because $p2$ is also a valid model that makes n *True*. Therefore, after this merging, we will have two sub-models for n being *True*: {{a/True, c/True, d/True, b/False}, {a/False, b/True}}.

Forward Propagation: Since nodes are assigned values is in a backward chaining manner, where we compute the truth assignment of the predecessors before that of the current node, the sub-models needs to cover as much information as possible. If some nodes' value can be inferred from the proven nodes, they must also be added into the sub-model. For example, suppose we have a sub-model {a/True, b/False} for making node n *True*. Suppose there are two additional rules related to node a and b: `(i) c :- a.` `(ii) d :- not b.` In this case, we know that c and d must also be *True*.

igASP propagates truth values every time a presumed node value has been established, by using a causal map which covers all of the causal relationships for each node/value. When a presumed node/value is established, igASP will check whether there is any other node whose value can be inferred from current

node assignments. If there are any, the inferred value is assigned to that node and propagation continues until the model does not change.

Query Handling: A query w.r.t. an ASP program amounts to checking whether a literal is in one of the models of the program. For instance, ASP program `p :- not q.  q :- not p.  :- p, q.` has two models {{p/True, q/False}, {p/False, q/True}}. If we query p, we should get the model $\{p\}$.

For query handling, igASP negates the query literal and append it to the ASP program as an additional constraint. So for the above example, the query $p/True$ will be converted to a constraint rule `:- not p.` and added to the original program. So the program will now be `p :- not q.  q :- not p.  :- p, q.  :- not p.`

Non-constrained (Non-headless-rules) Program Handling: igASP begins its reasoning from a constraint node (typically, the query represented as a constraint), then searches for a partial answer set to satisfy the constraint.

This may raise a concern: How about an ASP program that has no global constraints (headless rules) at all? To solve this problem, igASP performs a conversion on the original dependency graph.

Since all original facts in an ASP program should never be *False*. It means that we can take all negated facts as global constraints. Therefore, for any ASP program has default facts, we will generate global constraints accordingly. What if there is no fact in the program? In this case, igASP picks one node, and links it to the "Constraint" node with both positive and negative edges (via a conjunction node). The reason is simple, a node will either be *True* or *False*. For a program whose dependency graph is disconnected, igASP picks one node from each separated sub-graph, and links them to the "Constraint" node with both positive and negative edges. For picking which node to connect with the "Constraint" node, we use a heuristic which chooses the node with most in-coming edges. Since in-coming edges represent dependencies, and each sub-graph is connected, the heuristic is admissible.

4 Causal Justification

A major advantage of our graph approaches is that they provides justification as to why a literal is in an answer set for free. Providing justification is a major problem for implementations of ASP that are based on SAT solvers. In contrast to SAT-based ASP solvers, our graph representation maintains the information about structure of an ASP program while computing stable models. Indeed, the resulting graph itself is a justification tree. Since the truth values of all vertices are propagated along edges, we are able to find a justification by looking at the effective out-going edges and their ending nodes. Here the effective out-going edge refer to an edge that actually propagated *True* value to its ending node. According to propagation rules that are discussed in Section **??**, there are only two type of *effective* out-going edges: (i) positive edge coming from a *True* node; (ii) negative edge coming from a *False* node. Every effective out-going edge should point to a *True* node. Therefore, the justification first picks effective out-going edges, then check each edge's ending node. If all those ending nodes are *True*, the answer set is justified.

References

[1] Fang Li, Huaduo Wang & Gopal Gupta (2021): *grASP: A Graph Based ASP-Solver and Justification System*. *CoRR* abs/2104.01190. Available at `https://arxiv.org/abs/2104.01190`.

[2] Vladimir Lifschitz & Hudson Turner (1994): *Splitting a Logic Program*. In Pascal Van Hentenryck, editor: *Logic Programming, Proceedings of the Eleventh International Conference on Logic Programming, Santa Marherita Ligure, Italy, June 13-18, 1994*, MIT Press, pp. 23–37.

[3] Thomas Linke & Vladimir Sarsakov (2005): *Suitable graphs for answer set programming*. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, Springer, pp. 154–168, doi:10.1007/978-3-540-24609-1_26.

[4] Kyle Marple, Ajay Bansal, Richard Min & Gopal Gupta (2012): *Goal-directed execution of answer set programs*. In: *Proc. PPDP'12*, ACM, pp. 35–44, doi:10.1145/2370776.2370782.