A New Algorithm for Answer Set Computation

Giuliano Grossi and Massimo Marchi

Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
Via Comelico 39, I-20135 Milano, Italy

Abstract. A new exact algorithm for computing answer sets of logic programs is presented and analyzed. The algorithm takes a logic program in Kernel normal form as an input and computes its answer sets by reducing the problem to a suitable version of graph colorability. Even though worst-case complexity is exponential, thanks to a straightforward formulation we can prove that the algorithm works in time $\mathcal{O}^*(1.6181^n)$, which is asymptotically better than the trivial bound $\mathcal{O}^*(2^n)$ of the brute force algorithms. We argue that this new algorithm represents a significant progress in terms of worst-case time complexity over traditional branch-and-bound algorithms.

1 Introduction

This article introduces a new algorithm for computing the Answer Sets of a Logic Program (henceforth program) and studies its asymptotical properties. Our approach is based on two main concepts, which have been recently developed by Costantini et al. [1,2] in the context of checking existence of answer sets:

– the *kernel* normal form [2], by which logic programs are phrased as having only negative conditions, no undefined atoms and all atoms, in essence, are defined throughout negative cycles;
– the representation of programs as directed graphs with the *Extended Dependency Graph Encoding* [3], and of their answer sets in terms of *admissible* colorings of the EDG.

Focusing on Kernel programs and adopting the EDG encoding (as opposed to the traditional Dependency Graph) Costantini et al. were able to characterize the answer sets of a kernel program in terms of a class of 2-colorings called *admissible.* A coloring is admissible, in short, when i) no two adjacent nodes are both assigned to color *green* (interpreted as true) and ii) if a node is assigned to the dual *red* color then not all its in-neighbors are assigned to red as well. Admissible colorings represent answer sets modulo the mapping from nodes to the atoms they represent.

Those results make possible for us to consider Answer Set computation as an abstract graph-coloring problem and to attack it with traditional algorithms means. Our main result, therefore, is an algorithm that finds the admissible colorings of kernel graphs, i.e., graphs that are suitable to represent a kernel logic program. Our algorithm takes advantage of the admissibility conditions i) and ii) above to find total colorings that are admissible by construction.

In particular, this algorithm is not so much based on the idea of *propagating* coloring assignments through some sort of color alternation. On the opposite, the key point here is to come up with a suitable assignment of the green color, i.e., one that satisfies condition i) above, and only after that proceed to verify condition ii) on the red nodes. Promising assignments of the green color are obtained by reduction to the (equally complex) problem of finding maximal independent sets on the same graphs.

2 Preliminaries

We cannot here satisfactorily introduce the reader to Answer Set Programming, for which we invite the reader to refer to the excellent survey in [4]. However, as they are not standard we will now reproduce the definitions of Kernel program and of Extended Dependency Graph to which what follows will often refer.

Definition 1 (from [2]). *A logic program Π is in kernel normal form (or, equivalently, Π is a kernel program) if and only if the following conditions hold.*

1. *Π is WFS-irreducible, i.e., $WFS(\Pi) = \langle \varnothing, \varnothing \rangle$;*
2. *every rule has its body composed of negative literals only;*
3. *every atom in Π occurs in the body of some rule.*

It is easy to see that, in kernel programs, every atom occurs as the head of some rule. Of course, there are no facts. Moreover, one can observe that all atoms are either part of a cyclic definition or defined using atoms that are part of a cycle. In other words, all atoms *somewhat* depend on cycles. This notion is made precise and developed in the work of [1].

Every kernel program, and in general every logic program, can be rewritten in terms of directed finite labeled graph called *Extended Dependency Graph* (EDG). This representation captures in a plain form the essence of dependencies among rules[1]. A constructive definition of these graphs for the general class of LP can be found in [3]. Due to the properties explained above, we will use for the kernel program a more simple definition for EDG, called *Kernel EDG*. Under this definition, every kernel program can be expressed in terms of directed *unlabeled* finite graph which has the interesting property that each vertex has at least a incoming edge and at least an outgoing edge.

3 Graph-theory Preliminaries

A directed graph (*digraph*) is a pair $G = \langle V, E \rangle$ made up the vertex set $V = \{1, \ldots, n\}$ and an edge set $E \subseteq V^2$. If not otherwise stated, n will always refer to the number of vertexes in a graph and m will refer to the number of edges. For an edge $e = (i, j)$, the vertex i is called *initial endpoint* and the vertex j is called *terminal endpoint*.

We denote by $\mathcal{N}^+(i)$ the set of *in-neighbors*, i.e., the vertexes that are initial endpoints in all arcs in which i is terminal endpoint. Similarly, we denote by $\mathcal{N}^-(i)$ the set

[1] Another way to formalize the class of EDG is the Rule Dependency Graph (RDG), introduced in [5].

of *out-neighbors*, i.e., the vertexes that are terminal endpoints in all arcs in which i is initial endpoint. The set of all *neighbors* of the vertex i denoted by $\mathcal{N}(i)$ end defined as $\mathcal{N}(i) = \mathcal{N}^+(i) \cup \mathcal{N}^-(i)$.

The *underlying* graph $\hat{G} = \langle V, \hat{E} \rangle$ associated to a digraph $G = \langle V, E \rangle$ is an undirected graph that has the same set of vertexes V and the set of edges $\hat{E} = \{\{i,j\} : i \neq j \in V \wedge (i,j) \in E \vee (j,i) \in E\}$. Subsets of vertexes correspond to *induced* subgraphs of G, in which we include all edges between vertexes in the subset. We write $deg(i,S)$ to denote the degree of vertex i in the subgraph induced by S.

An *independent set* for an undirected graph $\hat{G}$ is a subset $S \subseteq V$ that does not induce any edges. It has been shown in [6] that a graph of n vertexes contains at most $3^{\frac{n}{3}} \approx 1.4422^n$ *maximal* (with respect to the inclusion) independent set (MIS).

If S and T are sets, $S \backslash T$ denotes the *set-theoretic* difference, consisting of elements of S that are not contained in T.

Throughout this paper, we will measure the running times of algorithms using the so called *big-Oh-star* notation that suppresses polynomially bounded terms. We write $\mathcal{O}^*(T(n))$ for a time complexity of the form $\mathcal{O}(T(n)\text{poly}(n))$, for some polynomial poly(n). This modification is justified by the exponential growth[2] of $T(n)$.

The algorithm will be presented using a C-style pseudo-code syntax. We assume the usual RAM model of computation, in which a single cell is capable of storing an integer large enough to index the memory requirements of the program, and in which arithmetic and array indexing operations on these values are assumed to take constant time.

4 From Answer Set Computation to Graph Coloring

In this Section we recall a reduction from ASP to a particular type of coloring of the relative EDGs [3], and we show that an admissible coloring is a MIS for that graph.

What is important in the definition of EDG is the careful distinction between rules that have the same heads. This is done by introducing indexes for rules defining the same atom, e.g.,

$$\begin{array}{l} p \leftarrow \textit{not}\ a. \\ p \leftarrow \textit{not}\ b. \\ \ldots \end{array}$$

will be denoted

$$\begin{array}{l} p^1 \leftarrow \textit{not}\ a. \\ p^2 \leftarrow \textit{not}\ b. \\ \ldots \end{array}$$

To focus on this essential aspect of the EDG representation we will reformulate the EDG representation for kernel programs.

[2] For instance, for graphs with n vertexes and m edges, the running time $1.7344^n n^2 m^5$ is sandwiched between the running times 1.7344^n and 1.7345^n.

Definition 2 (Kernel Extended Dependency Graph). *For a kernel program $\Pi_{\mathcal{K}}$, its Extended Dependency Graph EDG$(\Pi_{\mathcal{K}})$ (or simply EDG) is the digraph $\langle V, E\rangle$ defined as follows:*

1. *for each rule in $\Pi_{\mathcal{K}}$, $a_i^{(k)} \in V$, where a_i is the name of the head and k is the progressive index of the rules defining a_i;*
2. *for each $a_j \in V$, $(a_j, a_i^{(k)}) \in E$ if and only if a_j appears in the body of the k-th rule defining a_i.*

It is easy to check that EDGs are isomorphic to programs [3] modulo the mapping between nodes and vertexes. Since kernel programs have negative conditions only, in the rest of the article we will use the simplified notation $\langle V, E\rangle$ to denote an EDG.

Let us now introduce admissible 2-colorings of EDG and we show its interpretation in terms of answer sets of Π.

Definition 3 (RED-GREEN coloring). *A* R-G coloring *for the graph EDG is a total function $\rho : V \to \{\text{R}, \text{G}\}$. Moreover, it is* admissible *if the following two conditions hold:*

1. *if $\exists i$ such that $\rho(i) = \text{G}$, then $\forall j \in \mathcal{N}(i)$ $\rho(j) = \text{R}$;*
2. *if $\exists i$ such that $\rho(i) = \text{R}$, then $\exists j \in \mathcal{N}^+(i)$ such that $\rho(j) = \text{G}$.*

In other words, the two conditions given in the previous definition mean:

1. no two green vertexes are adjacent (*violation of type* G-G);
2. no red vertex has all its in-neighbors red (*violation of type* R-R).

The main result that makes EDG an appealing representation of ASP programs is the isomorphism between admissible EDG colorings and answer sets. For any interpretation $S \subseteq$ of Π, an **associate coloring** col_S is a function that satisfies the condition: $a_i \in S$ if and only if $\exists k.col_S(a_i^{(k)}) = \text{G}$. Clearly, more than one coloring can be associated to S, but for the moment we won't discuss this issue.

Theorem 1 (from [3]). *An interpretation S is an answer set of Π if and only if there is an associated coloring col_S which is admissible for EDG.*

Example 1 (Logic program-EDG association). Let us consider the following logic program Π:

$$\begin{aligned} p &\leftarrow \textit{not}\ a.\\ p &\leftarrow \textit{not}\ b.\\ a &\leftarrow \textit{not}\ b.\\ b &\leftarrow \textit{not}\ a. \end{aligned}$$

Clearly, there are two stable models, $S_1 = \{p, a\}$ and $S_2 = \{p, b\}$.

The EDG associated to Π (see Fig.1) has four vertexes, $\{p, p', a, b\}$ and four arcs $\{(a, p), (b, p'), (b, a), (a, b)\}$. There are two admissible colorings, i.e., $\rho(p) = \rho(b) = \text{G}$ and $\rho(p') = \rho(b) = \text{G}$ (everything else being mapped on R) associated to S_1 and S_2, respectively.

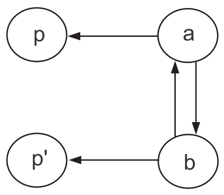

Fig. 1. The EDG associated to Π

An important property of the coloring defined above and useful to derive the recursive algorithm presented in the next section is given in the following

Proposition 1. *Let $\rho : V \rightarrow \{\text{R}, \text{G}\}$ be an* R-G *coloring for $EDG = \langle V, E \rangle$, and $V_G \subseteq V$ be the subset of green vertexes of EDG. If ρ is admissible then V_G is a maximal independent set for the underlying graph $\hat{U}$ of EDG.*

Proof. First of all, we show that if ρ is an admissible R-G coloring then V_G is an independent set for the underlying graph $\hat{U}$ of EDG.

The fact that the vertexes in V_G are pairwise independent is given directly by the definition of admissible R-G coloring, for which two green adjacent vertexes are not allowed.

To show that it is also maximal, let us suppose, by absurd, that V_G is not maximal, i.e., there exist a vertex i in $V \setminus V_G$ (red colored) that is not adjacent to all vertexes of V_G. This implies that i has not a green vertex between its in-neighbors, leading to a R-R violation and then a contradiction for the admissible coloring. □

5 The Algorithm for EDG Coloring

We now introduce our algorithm. Even though it has exponential worst-case complexity, we believe that it is promising to become significantly faster than traditional $\mathcal{O}^*(2^n)$ exhaustive search.

The idea is essentially based on a branch-and-bound technique. At each step a partial solution is adjusted by selecting a vertex not yet colored and trying both the colors (RED end GREEN) into two different branches of execution. This naturally defines a search tree: every branching propagates the search into subtree recursively. Each subtree can be associated to a partial coloring of the vertexes.

Sometimes, it can be seen that a particular subtree leads to an infeasible solution because the associated partial coloring rise a violation. In this case the subtree can be safely abandoned thus cutting off the search for this branch. Of course, pruning the search tree speeds-up the entire process.

A nice consequence of our pruning method is the mathematical understanding of the evolution of the search tree in terms of subtree pruned and the remaining subtree. This approach permits to easily compute the worst case time complexity of the algorithm at hand.

Given a digraph $G = \langle V, E\rangle$, we are ready to describe an algorithm essentially built on a recursive procedure on subgraphs induced by a partial coloring, as explained above. In fact, its search direction has two aims:

1. to guide toward all MIS contained in the underlying graph $\hat{G}$ representing admissible R-G coloring, as motivated in Proposition 1;
2. to prune the subtree when a violation R-R is encountered.

The idea is to branch on a proper vertex (for instance those of high degree): If the vertexes contained in the subgraph induced by the not yet colored vertexes have all degree zero, then they can be all GREEN colored because all neighbors (yet colored) are RED, and then to check whether the coloring is admissible. Otherwise, if G contains at least a vertex i of degree at least one, then every admissible coloring falls into one of the following classes:

1. i is RED colored;
2. i is GREEN colored, then all of its neighbors must be RED colored.

We now branch the computation into two subtrees. The first subtree deals with the graph that results from RED coloring the vertex i in G. The second subtree deals with the graph that results from GREEN coloring i together with all neighbors RED colored in G. We recursively compute the MIS in both subtrees, and update it to a solution for the original graph G.

Let us observe that in this way a G-G violation will never occur, since when a vertex is green colored all neighbors are automatically red colored, and thus preserving the independence of the green vertexes.

A more detailed pseudo-code description of the algorithm is given in Listing 1.1. In particular, it is detailed the recursive procedure EDG_COLORING, while the procedure FINAL_CHECK is only mentioned and not explained because it has a trivial task: to assign all vertexes not yet colored to the MIS and check for admissible coloring.

It is easy to see that the procedure FINAL_CHECK has polynomial time complexity, since it is running at the end of the recursive procedure when all the remaining vertexes have degree zero in the subgraph induced by the not yet colored vertexes.

As far as the worst case time complexity of the entire algorithm is concerned, we can conclude that

Theorem 2. *We can compute all* R-G *colorings of EDG in time $\mathcal{O}^*(1.6181^n)$.*

Proof. Denote by $T(n)$ the worst case time that this algorithm needs on a graph with n vertexes. Then,

$$T(n) \leq T(n-2) + T(n-1) + \mathcal{O}(n+m)$$

because the procedure applies recursively a search on two subgraphs which are reduced by of at least 2 and exactly 1 vertex respectively.

Standard calculations yield that $T(n)$ is within a polynomial factor of α^n where 1.6181 is the largest real root of $\alpha^2 = \alpha + 1$. This yields the time complexity $\mathcal{O}^*(1.6181^n)$. □

Listing 1.1. Recursive procedure EDG_COLORING

```
/*
  – The procedure takes an EDG graph $G = \langle V, E\rangle$ as input and
    list all its admissible coloring.
  – At each step $V = C \cup \bar{C}$, where $C$ and $\bar{C} = V \setminus C$ represent
    the set of already colored vertexes and the set of
    vertexes to be colored, respectively.
  – $i_G$ and $i_R$ means that vertex $i$ is GREEN or RED colored,
    respectively.
*/

void EDG_COLORING(digraph $G = \langle C \cup \bar{C}, E\rangle$) {

   // check whether there are vertexes not yet colored
   if (there exists $i \in \bar{C}$ with $deg(i, \bar{C}) \geq 1$)
      i = choose_a_vertex_to_color($\bar{C}$);
   else
      FINAL_CHECK($G$);

   // check whether GREEN coloring vertex $i$ induces
   // an R–R violation on the neighbors of $i$
   if (RED coloring all $j \in \mathcal{N}(i)$ doesn't induce R-R viol.)
      EDG_COLORING($\langle C \cup \{i_G\} \cup \{j_R : j \in \mathcal{N}(i)\} \cup \bar{C}, E\rangle$);

   // check if RED coloring the vertex $i$ causes a violation
   // otherwise calls the procedure on vertex $i$ RED colored
   if (RED coloring $i$ induces an R-R violation)
      return; // remove from the stack
   else
      EDG_COLORING($\langle C \cup \{i_R\} \cup \bar{C}, E\rangle$);
}
```

6 Comparison with Literature

Perhaps the work in literature closest in spirit to ours is the excellent study in [7] over the worst-case analysis of Answer Set computation. The main result there consists of an algorithm for Answer set computation that works in $\mathcal{O}^*(m \times 1.44225^n)$ where *m* is the size of the input program and *n* the number of atoms thereof. Such result is given for a class of syntax-restricted programs that are somewhat orthogonal to that we consider. Therefore, we are not able at the moment to propose a complete comparison between the two methods.

A similar graph coloring technique for solving LP is presented in [5] and used into system *noMoRe*. Although slightly similar in the data structures, which imply a comparable space complexity, it differs from our approach in the way to realize the total coloring. In [5] the expansion is based on a group of set-transformations which use the concept of support graph. The work doesn't threat the time-complexity so we cannot make any comparison with it.

7 Conclusions

We have proposed an exact algorithm for Answer Set computation that shows promising asymptotic worst-case complexity.

Our algorithm is designed to work on the EDG graph representation of programs. Since the EDG is by design oriented to represent the rules of the program explicitly, an issue that needs further study is the trade-off between the number of rules and the worst-case complexity. So far, we have only anecdotal evidence that the number of rules of a program depends only linearly on the number of its atoms. When this condition is the case, a significant fact rises from the worst case analysis: to the best of our knowledge our algorithm is the first exhaustive algorithm which provably breaks the complexity $\mathcal{O}^*(2^n)$ of the exhaustive search.

References

1. Costantini, S.: On the existence of stable models of non-stratified logic programs. Theory and Practice of Logic Programming, accepted for publication (2005)
2. Costantini, S., Provetti, A.: Normal forms for answer set programming. Theory and Practice of Logic Programming, accepted for publication (2005)
3. Costantini, S., D'Antona, O., Provetti, A.: On the equivalence and range of applicability of graph-based representations of logic programs. Information Processing Letters 84(5) (2002) 241–249
4. Marek, W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. The Logic Programming Paradigm: a 25-Year Perspective Springer-Verlag (1999) 375–398
5. Konczak, K., Schaub, T., Linke, T.: Graphs and colorings for answer set programming: Abridged report. In: Answer Set Programming. (2003)
6. Moon, J.W., Moser, L.: On cliques in graphs. Israel Journal of Mathematics 3 (1965) 23–28
7. Lonc, Z., Truszczynski, M.: Computing stable models: Worst-case performance estimates. In: ICLP '02: Proceedings of the 18th International Conference on Logic Programming, Springer-Verlag LNCS (2002) 347–362