On the implementation of a multiple outputs algorithm for defeasible argumentation

Teresa Alsinet[1], Ramón Béjar[1], Lluis Godo[2], and Francesc Guitart[1]

[1] Department of Computer Science – University of Lleida
Jaume II, 69 – 25001 Lleida, SPAIN
{tracy, ramon, fguitart}@diei.udl.cat
[2] Artificial Intelligence Research Institute (IIIA-CSIC)
Campus UAB - 08193 Bellaterra, Barcelona, SPAIN
godo@iiia.csic.es

Abstract. In a previous work we defined a recursive warrant semantics for Defeasible Logic Programming based on a general notion of collective conflict among arguments. The main feature of this recursive semantics is that an output of a program is a pair consisting of a set of warranted and a set of blocked formulas. A program may have multiple outputs in case of circular definitions of conflicts among arguments. In this paper we design an algorithm for computing each output and we provide an experimental evaluation of the algorithm based on two SAT encodings defined for the two main combinatorial subproblems that arise when computing warranted and blocked conclusions for each output.

1 Introduction and motivation

Defeasible Logic Programming (DeLP) [8] is a formalism that combines techniques of both logic programming and defeasible argumentation. As in logic programming, knowledge is represented in DeLP using facts and rules; however, DeLP also provides the possibility of representing defeasible knowledge under the form of weak (defeasible) rules, expressing reasons to believe in a given conclusion. In DeLP, a conclusion succeeds in a program if it is warranted, i.e., if there exists an argument (a consistent set of defeasible rules) that, together with non-defeasible rules and facts, entails the conclusion, and moreover, this argument is found to be undefeated by a warrant procedure. This builds a dialectical tree containing all arguments that challenge this argument, and all counterarguments that challenge those arguments, and so on, recursively. Actually, dialectical trees systematically explore the universe of arguments in order to present an exhaustive synthesis of the relevant chains of pros and cons for a given conclusion.

In [1] we defined a new recursive semantics for DeLP based on a general notion of collective (non-binary) conflict among arguments. In this framework, called *Recursive* DeLP (R-DeLP for short), an output (or extension) of a program is a pair consisting of a set of warranted and a set of blocked formulas. Arguments for both warranted and blocked formulas are recursively based on warranted formulas but, while warranted formulas do not generate any collective conflict, blocked conclusions do. Formulas that are neither warranted nor blocked correspond to rejected formulas. The key feature that our warrant recursive semantics addresses is the *closure under subarguments postulate* recently proposed by Amgoud[4], claiming that if an argument is excluded from an output, then all the arguments built on top of it should also be excluded from that output.

Then, in case of circular definitions of conflict among arguments, the recursive semantics for warranted conclusions may result in multiple outputs for R-DeLP programs.

In this paper, after overviewing in Section 2 the main elements of the warrant recursive semantics for R-DeLP, in Section 3 we design an algorithm for computing every output for R-DeLP programs with multiple outputs, and in Section 4 we present empirical results. These are obtained with an implementation of the algorithm based on two SAT encodings defined in [2] for the two main combinatorial subproblems that arise when computing warranted and blocked conclusions for each output for an R-DeLP program, so that we can take profit of existing state-of-the-art SAT solvers for solving instances of big size.

2 Preliminaries on R-DeLP

The *language* of R-DeLP, denoted $\mathcal{L}$, is inherited from the language of logic programming, including the notions of atom, literal, rule and fact. Formulas are built over a finite set of propositional variables $\{p, q, \ldots\}$ which is extended with a new (negated) atom "$\sim p$" for each original atom p. Atoms of the form p or $\sim p$ will be referred as literals.[3] *Formulas* of $\mathcal{L}$ consist of rules of the form $Q \leftarrow P_1 \wedge \ldots \wedge P_k$, where $Q, P_1, \ldots, P_k$ are literals. A fact will be a rule with no premises. We will also use the name *clause* to denote a rule or a fact. The R-DeLP framework is based on the propositional logic $(\mathcal{L}, \vdash)$ where the inference operator $\vdash$ is defined by instances of the modus ponens rule of the form: $\{Q \leftarrow P_1 \wedge \ldots \wedge P_k, P_1, \ldots, P_k\} \vdash Q$. A set of clauses Γ will be deemed as *contradictory*, denoted $\Gamma \vdash \bot$, if , for some atom q, $\Gamma \vdash q$ and $\Gamma \vdash \sim q$.

An R-DeLP *program* $\mathcal{P}$ is a tuple $\mathcal{P} = (\Pi, \Delta)$ over the logic $(\mathcal{L}, \vdash)$, where $\Pi, \Delta \subseteq \mathcal{L}$, and $\Pi \nvdash \bot$. Π is a finite set of clauses representing strict knowledge (information we take for granted they hold true), Δ is another finite set of clauses representing the defeasible knowledge (formulas for which we have reasons to believe they are true).

The notion of *argument* is the usual one. Given an R-DeLP program $\mathcal{P}$, an argument for a literal (conclusion) Q of $\mathcal{L}$ is a pair $\mathcal{A} = \langle A, Q \rangle$, with $A \subseteq \Delta$ such that $\Pi \cup A \nvdash_R \bot$, and A is minimal (with respect to set inclusion) such that $\Pi \cup A \vdash Q$. If $A = \emptyset$, then we will call $\mathcal{A}$ a s-argument (s for strict), otherwise it will be a d-argument (d for defeasible). The notion of *subargument* is referred to d-arguments and expresses an incremental proof relationship between arguments which is defined as follows. Let $\langle B, Q \rangle$ and $\langle A, P \rangle$ be two d-arguments such that the minimal sets (with respect to set inclusion) $\Pi_Q \subseteq \Pi$ and $\Pi_P \subseteq \Pi$ such that $\Pi_Q \cup B \vdash_R Q$ and $\Pi_P \cup A \vdash_R P$ verify that $\Pi_Q \subseteq \Pi_P$. Then, $\langle B, Q \rangle$ is a *subargument* of $\langle A, P \rangle$, written $\langle B, Q \rangle \sqsubset \langle A, P \rangle$, when either $B \subset A$ (strict inclusion for defeasible knowledge), or $B = A$ and $\Pi_Q \subset \Pi_A$ (strict inclusion for strict knowledge). More generally, we say that $\langle B, Q \rangle$ is a *subargument of a set of arguments* G, written $\langle B, Q \rangle \sqsubset G$, if $\langle B, Q \rangle \sqsubset \langle A, P \rangle$ for some $\langle A, P \rangle \in G$. A literal Q of $\mathcal{L}$ is called *justifiable conclusion* with respect to $\mathcal{P}$ if there exists an argument for Q, i.e. there exists $A \subseteq \Delta$ such that $\langle A, Q \rangle$ is an argument.

The warrant recursive semantics for R-DeLP is based on the following notion of collective conflict. Let $\mathcal{P} = (\Pi, \Delta)$ be an R-DeLP program and let $W \subseteq \mathcal{L}$ be a set of conclusions. We say that a set of arguments $\{\langle A_1, Q_1 \rangle, \ldots, \langle A_k, Q_k \rangle\}$ *minimally*

[3] For a given literal Q, we write $\sim Q$ as an abbreviation to denote "$\sim q$" if $Q = q$ and "q" if $Q = \sim q$.

conflicts with respect to W iff the two following conditions hold: (i) the set of argument conclusions $\{Q_1, \ldots, Q_k\}$ is contradictory with respect to W, i.e. it holds that $\Pi \cup W \cup \{Q_1, \ldots, Q_k\} \vdash \bot$; and (ii) the set $\{\langle A_1, Q_1\rangle, \ldots, \langle A_k, Q_k\rangle\}$ is minimal with respect to set inclusion satisfying (i), i.e. if $S \subsetneq \{Q_1, \ldots, Q_k\}$, then $\Pi \cup W \cup S \nvdash \bot$.

An *output for an R-DeLP program* $\mathcal{P} = (\Pi, \Delta)$ is any pair $(\mathit{Warr}, \mathit{Block})$, where $\mathit{Warr} \cap \mathit{Block} = \emptyset$ and $\{Q \mid \Pi \vdash Q\} \subseteq \mathit{Warr}$, satisfying the following recursive constraints:

1. $P \in \mathit{Warr} \cup \mathit{Block}$ iff there exists an argument $\langle A, P\rangle$ such that for every $\langle B, Q\rangle \sqsubset \langle A, P\rangle$, $Q \in \mathit{Warr}$. In this case we say that the argument $\langle A, Q\rangle$ is *valid* with respect to *Warr*.
2. For each valid argument $\langle A, Q\rangle$:
   - $Q \in \mathit{Block}$ whenever there exists a set of valid arguments G such that
     (i) $\langle A, Q\rangle \not\sqsubset G$, and
     (ii) $\{\langle A, Q\rangle\} \cup G$ minimally conflicts with respect to the set $W = \{P \mid \langle B, P\rangle \sqsubset G \cup \{\langle A, Q\rangle\}\}$.
   - otherwise, $Q \in \mathit{Warr}$.

In [1] we showed that, in case of some circular definitions of conflict among arguments, the output of an R-DeLP program may be not unique, that is, there may exist several pairs $(\mathit{Warr}, \mathit{Block})$ satisfying the above conditions for a given R-DeLP program. Following the approach of Pollock [9], circular definitions of conflict were formalized by means of what we called *warrant dependency graphs*. A warrant dependency graph represents (i) support relations of almost valid arguments with respect to valid arguments and (ii) conflict relations of valid arguments with respect to almost valid arguments. An almost valid argument is an argument based on a set of valid arguments and whose status is warranted or blocked (but not rejected), whenever every valid argument in the set is warranted, and rejected, otherwise. Then, a cycle in a warrant dependency graph represents a circular definition of conflict among a set of arguments.

3 Computing the set of outputs for an R-DeLP program

From a computational point of view, an output for an R-DeLP program can be computed by means of a recursive procedure, starting with the computation of warranted conclusions from strict clauses and recursively going from warranted conclusions to defeasible arguments based on them. Next we design an algorithm implementing this procedure for computing warranted and blocked conclusions by checking the existence of conflicts between valid arguments and cycles at some warrant dependency graph.

The algorithm `R-DeLP outputs` first computes the set of warranted conclusions form the set of strict clauses Π. Then, computes the set *VA* of valid arguments with respect to the strict part, i.e. arguments with an empty set of subarguments. The recursive procedure `extension` receives as input the current partially computed output (W, B) and the set of valid arguments *VA* and dynamically updates the set *VA* depending on new warranted and blocked conclusions and the appearance of cycles in some warrant dependence graph. When a cycle is found in a warrant dependence graph, each valid argument of the cycle can lead to a different output. Then, the procedure `extension` selects one valid argument of the cycle and recursively computes the resulting output by

warranting the selected argument. The procedure `extension` finishes when the status for every valid argument of the current output is computed. When an R-DeLP program has multiple outputs, each output is stored in the set of outputs O.

Algorithm `R-DeLP outputs`
Input $\mathcal{P} = (\Pi, \Delta)$: An R-DeLP program
Output O: Set of outputs for $\mathcal{P}$
Variables
(W, B): Current output for $\mathcal{P}$
VA: Set of valid arguments w.r.t. the current set of warranted conclusions W
Method
$O := \emptyset$
$W := \{Q \mid \Pi \vdash Q\}$
$B := \emptyset$
VA : = $\{\langle A, Q\rangle \mid \langle A, Q\rangle$ is valid w.r.t. $W\}$
`extension` $((W, B)$, *VA*, $O)$
end algorithm `R-DeLP outputs`

Procedure `extension` (**in** (W, B), *VA*; **in_out** O)
Variables
W_{ext}: Extended set of warranted conclusions
VA_{ext}: Extended set of valid arguments
is_leaf: Boolean
Method
is_leaf := `true`
while (*VA* $\neq \emptyset$ **and** *is_leaf* = `true`) **do**
while $(\exists\langle A, Q\rangle \in$ *VA* $\mid$
$\neg$ `conflict`$(\langle A, Q\rangle$, *VA*, W, `not_dependent`$(\langle A, Q\rangle$, `almost_valid`(*VA*, $(W, B))))$
and $\neg$ `cycle`$(\langle A, Q\rangle$, *VA*, W, `almost_valid`(*VA*, $(W, B)))$ **do**
$W := W \cup \{Q\}$
VA := *VA*$\backslash\{\langle A, Q\rangle\} \cup \{\langle C, P\rangle \mid \langle C, P\rangle$ is valid w.r.t. $W\}$
end while
$I := \{\langle A, Q\rangle \in$ *VA* $\mid$ `conflict`$(\langle A, Q\rangle$, *VA*, $W, \emptyset)\}$
$B := B \cup \{Q \mid \langle A, Q\rangle \in I\}$
VA := *VA*$\backslash I$
$J := \{\langle A, Q\rangle \in$ *VA* $\mid$ `cycle`$(\langle A, Q\rangle$, *VA*, W, `almost_valid`(*VA*, $(W, B)))\}$
for each argument $(\langle A, Q\rangle \in J)$ **do**
$W_{ext} := W \cup \{Q\}$
VA_{ext} := *VA*$\backslash\{\langle A, Q\rangle\} \cup \{\langle C, P\rangle \mid \langle C, P\rangle$ is valid w.r.t. $W_{ext}\}$
`extension` $((W_{ext}, B)$, VA_{ext}, $O)$
end for
if $(J \neq \emptyset)$ **then** *is_leaf* := `false`
end while
if $((W, B) \notin O$ **and** *is_leaf* = `true`) **then** $O := O \cup \{(W, B)\}$
end procedure `extension`

The function `almost_valid` computes the set of almost valid arguments based on some valid arguments in *VA*. The function `not_dependent` computes the set of almost valid arguments which do not depend on $\langle A, Q\rangle$. The function `conflict` has two different functionalities. On the one hand, the function `conflict` checks conflicts among the argument $\langle A, Q\rangle$ and the set *VA* of valid arguments, and thus, every valid argument involved in a conflict is blocked. On the other had, the function `conflict`

checks possible conflicts among the argument $\langle A, Q\rangle$ and the set *VA* of valid arguments extended with the set of almost valid arguments whose supports depend on some argument in *VA*$\backslash\{\langle A, Q\rangle\}$, and thus, every valid argument with options to be involved in a conflict remains as valid. Finally, the function `cycle` checks the existence of a cycle in the warrant dependency graph for the set of valid arguments *VA* and the set of almost valid arguments based on some valid arguments in *VA*.

One of the main advantages of the warrant recursive semantics is from the implementation point of view. Warrant semantics based on dialectical trees, like DeLP [5,6], might consider an exponential number of arguments with respect to the number of rules of a given program. However, for every output, our algorithm can be implemented to work in polynomial space. This can be achieved because it is not actually necessary to find all the valid arguments for a given literal Q, but only one witnessing a valid argument for Q is enough. Analogously, function `not_dependent` can be implemented to generate at most one almost valid argument not based on $\langle A, Q\rangle$ for a given literal. The only function that in the worst case can need an exponential number of arguments is `cycle`, but we showed [2] that whenever `cycle` returns true for $\langle A, Q\rangle$, then a conflict will be detected with the set of almost valid arguments which do not depend on $\langle A, Q\rangle$. Moreover, the set of valid arguments J computed by function `cycle` can also be computed by checking the stability of the set of valid arguments after two consecutive iterations, so it is not necessary to explicitly compute dependency graphs.

4 Empirical results

The algorithm `R-DeLP outputs` needs to compute two main queries during its execution: i) whether an argument is almost valid and ii) whether there is a conflict for a valid argument. In [2] we proposed SAT encodings for resolving both queries with a SAT solver.

Now in this paper we study the average number of outputs for R-DeLP instances and the median computational cost of solving them with the `R-DeLP outputs` algorithm, as the instances size increase with different instances characteristics. The main algorithm has been implemented with python, and for solving the SAT encodings, we have used the solver MiniSAT [7]. An on-line web based implementation of the R-DeLP argumentation framework is available at the URL: http://arinf.udl.cat/rp-delp.

To generate R-DeLP problem instances with different sizes and characteristics, we have used the generator algorithm described in [3]. We generate test-sets of instances with different number of variables (V): $\{15, 20, 25, 30\}$[4] and with clauses with one or two literals. For each number of variables, we generate three sets of instances, each one with a different ratio of clauses to variables (C/V): $\{2, 4, 6\}$. From all the clauses of an instance, a 10% of them are considered in the strict part of the program (Π) and a 90% of them are considered in the defeasible part (Δ). [5]

Table 1 shows the experimental results obtained for our test-sets. So far, we have computed the average number of outputs per instance ($\#$ O), the average number of

[4] Notice that the total number of literals is two times the number of variables.

[5] These parameters are selected given the experimental results in [3], where we considered single output programs and these parameters gave non-trivial instances in the sense that they are computationally hard to solve.

warrants per output, the average number of warrants in the intersection of the set of outputs and the median time for solving the instances. The results show that even for a small number of variables V and a small ratio C/V we can have instances with multiple outputs. Observe that although the average number of outputs is not too different between all the test-sets, the complexity of solving the instances seems to increase exponentially as either V or C/V increases. We believe this is mainly due to an increase in the complexity of deciding the final status (warranted or blocked) of each literal for each output.

V	C/V	$\#\ O$	$\#$ Warrants per output	$\#$ Warrants in intersection	Time (s.)
15	2	1.04	7.14	7.1	0.906
	4	1.31	7.28	6.93	7.96
	6	5.40	6.43	5.90	19.11
20	2	1.06	9.89	9.82	1.93
	4	1.65	9.63	9.31	28.89
	6	1.44	9.09	8.81	38.87
25	2	1.10	11.47	11.38	5.09
	4	2.44	11.72	10.74	76.31
	6	1.90	8.04	7.50	151.31
30	2	1.08	12.56	12.5	9.20
	4	1.81	12.16	11.56	142.49
	6	1.89	11.02	9.92	227.97

Table 1. Experimental results for the `R-DeLP outputs` algorithm.

As future work we propose to study the average number of blocked conclusions per output which would give us an idea if the computation time is higher because there is a relationship between the number of blocked and warranted conclusions. We also propose to extend the `R-DeLP outputs` algorithm to the case of multiple levels for defeasible facts and rules.

References

1. T. Alsinet, R. Béjar, and L. Godo. A characterization of collective conflict for defeasible argumentation. In *COMMA 2010*, pages 27–38.
2. T. Alsinet, R. Béjar, L. Godo, and F. Guitart. Maximal ideal recursive semantics for defeasible argumentation. In *SUM 2011*, LNAI 6929, pages 96–109.
3. T. Alsinet, R. Béjar, L. Godo, and F. Guitart. Using answer set programming for an scalable implementation of defeasible argumentation. In *ICTAI 2012*, pages 1016-1021.
4. L. Amgoud. Postulates for logic-based argumentation systems. In *ECAI 2012 Workshop WL4AI*, pages 59–67.
5. L. Cecchi, P. Fillottrani, and G. Simari. On the complexity of DeLP through game semantics. In *NMR 2006*, pages 386–394.
6. C.I. Chesñevar, G.R. Simari, and L. Godo. Computing dialectical trees efficiently in possibilistic defeasible logic programming. In *LPNMR 2005*, LNAI 3662, pages 158–171.
7. N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT 2003*, pages 502–518.
8. A. García and G.R. Simari. Defeasible Logic Programming: An Argumentative Approach. *Theory and Practice of Logic Programming*, 4(1):95–138, 2004.
9. John L. Pollock. A recursive semantics for defeasible reasoning. In Rahwan and Simari (eds.) *Argumentation in Artificial Intelligence*, pages 173–198. Springer, 2009.