Building precompiled knowledge in ODeLP[*]

Marcela Capobianco
Artificial Intelligence Research and Development Laboratory
Department of Computer Science and Engineering
Universidad Nacional del Sur – Av. Alem 1253, (8000) Bahía Blanca, ARGENTINA
EMAIL: `mc@cs.uns.edu.ar`

and

Guillermo R. Simari
Artificial Intelligence Research and Development Laboratory
Department of Computer Science and Engineering
Universidad Nacional del Sur – Av. Alem 1253, (8000) Bahía Blanca, ARGENTINA
EMAIL: `grs@cs.uns.edu.ar`

Abstract

Argumentation systems have substantially evolved in the past few years, resulting in adequate tools to model some forms of common sense reasoning. This has sprung a new set of argument-based applications in diverse areas.

In previous work, we defined how to use precompiled knowledge to obtain significant speed-ups in the inference process of an argument-based system. This development is based on a logic programming system with an argumentation-driven inference engine, called Observation Based Defeasible Logic Programming (ODeLP). In this setting was first presented the concept of *dialectical databases*, that is, data structures for storing precompiled knowledge. These structures provide precompiled information about inferences and can be used to speed up the inference process, as TMS do in general problem solvers.

In this work, we present detailed algorithms for the creation of dialectical databases in ODeLP and analyze these algorithms in terms of their computational complexity.

Keywords: Non-monotonic reasoning, Argumentation, Computational complexity

1 INTRODUCTION

Argumentation systems have substantially evolved in the past few years, resulting in adequate tools to model some forms of common sense reasoning. This has sprung a new set of argument-based applications in diverse areas, where knowledge representation issues play a major role, such as clustering algorithms [13], intelligent web search [6] and critiquing systems [5].

In previous work [3], we defined how to use precompiled knowledge to obtain significant speed-ups in the inference process of an argument-based system. The development is based on a logic programming system that uses an argumentation driven inference engine, called Observation Based

[*]This work was partially supported by Agencia Nacional de Promoción Científica y Tecnológica (PAV 2003 Nro. 076, PICT 2002 Nro. 13096, PICT 2003 Nro. 15043), and CONICET (*Consejo Nacional de Investigaciones Científicas y Técnicas de la República Argentina*).

Defeasible Logic Programming (ODeLP). Logic programming approaches to argumentation [7, 16] have proved to be suitable formalization tools in different application domains as they combine the powerful features provided by logic programming for knowledge representation together with the ability to model complex, argument-based inference procedures in unified, integrated frameworks.

In these models, real time issues play a particularly important role when modeling most applications, specially those concerning interactive systems. In argument-based approaches a timely interaction is especially hard to achieve, as the inference process involved is complex and computationally expensive. To achieve this kind of interaction we proposed the use of precompiled knowledge for argumentation systems, in the same way *truth maintenance systems* (TMS) [9] use precompiled knowledge to improve the performance of problem solvers.

To implement this idea we defined in [3] the concept of *dialectical databases*. These are data structures that store precompiled knowledge, providing precompiled information about inferences that can be used to speed up the inference process, as TMS do in general problem solvers. We discussed the main issues of the integration of dialectical databases in ODeLP, such as defining the theoretical background and modifying the inference process to take advantage of the new component.

In this work, we present detailed algorithms for the creation of dialectical databases in ODeLP. Then, we analyze these algorithms in terms of their computational complexity. The remainder of this paper is organized as follows. First, we present a brief overview of the ODeLP system. Next, we detail the rol of dialectical databases as structures of precompiled knowledge to assist inference, and finally we formulate and analyze the algorithms for dialectical databases creation in ODeLP.

2 RELATED WORK

Before addressing the contributions of our work, we present a brief overview of related work in the fields of precompiled knowledge. In *truth maintenance systems* (TMS) the use of precompiled knowledge helps improve the performance of problem solvers. A similar technique will be used in ODeLP to address real time constrains.

Truth Maintenance Systems (TMS) were defined by Doyle in [9] as support tools for problems solvers. The function of a TMS is to record and maintain the reasons for an agent's beliefs. Doyle describes a series of procedures that determine the current set of beliefs and update it in accord with new incoming reasons. Under this view, *rational thought* is deemed as the process of finding reasons for attitudes [9]. Some attitude (such as belief, desire, etc.) is rational if it is supported by some acceptable explanation.

Basically, TMS have two basic data structures: *nodes*, which represent beliefs, and *justifications* which model reasons for the nodes. The TMS believes in a node if it has a justification for the node and believes in the nodes involved in it. Although this may seem circular, there are assumptions (a special type of justifications) which involve no other nodes. Justifications for nodes may be added or retracted, and this accounts for a *truth maintenance procedure* [9], to make any necessary revisions in the set of beliefs. An interesting feature of TMS is the use of a particular type of justifications, called *non-monotonic*, to make tentative guesses. A non-monotonic justification bases an argument for a node not only on current beliefs in certain nodes, but also on lack of beliefs in other nodes. Any node supported by a non-monotonic justification is called an *assumption*.

TMS solve part of the belief revision problem in general problem solvers and provide a mechanism for making non-monotonic assumptions. As Doyle mentions in [9] performance is also significantly improved, even though the overhead required to record justifications for every program belief might seem excessive, we must consider the expense of not keeping these records. When information about derivations is discarded, the same information must be continually re-derived, even when only irrelevant assumptions have changed.

The fundamental actions of a TMS are create or retract nodes (to which the problem solving program using the TMS can attach the statement of a belief) and add (or retract) a justification for a node, to represent a step of an argument for the belief represented by the node. The system can also mark a node as a contradiction, to represent the inconsistency of any set of beliefs which enters into an argument for the node.

Every node in the TMS has an associated set of justifications. Each justification represents a different reason for asserting it. The node is believed if and only if at least one of the justifications is *valid*.[1] In this case it is say to be *in* the set of beliefs. Otherwise, the node is *out* of this set. In the TMS, each potential belief to be used as a hypothesis or a conclusion of an argument must be given its own distinct node. When uncertainty about some inference P exists, nodes for both P and its negation must be provided. Either of these nodes can have or lack well-founded arguments, leading to a four-element belief set (neither P nor ~P are believed, exactly one is believed, or both are believed). The author details the procedures needed to establish the state of every node, and to update these states in case new justifications or facts are added to the TMS.

Since the appearance of TMS a large body of literature and applications have been developed [8, 10, 15, 11, 2]. The original idea appears not to have been any particular technical mechanism, but the general concept of an independent module for belief maintenance [15].

In this and previous work [3] we apply the same idea to argument systems. This is a novel idea in the argumentation field and has not been introduced before in any argumentation framework.

3 ODELP: OBSERVATION-BASED DELP

Observation based Defeasible Logic Programming (ODeLP) [3] is a language for knowledge representation and reasoning that uses *defeasible argumentation* to decide between contradictory conclusions through a *dialectical analysis*. ODeLP can be seen as an specialization of the DeLP language [12] useful for dynamic environments, since it provides perception mechanisms to incorporate the changes in the world and integrate them into the knowledge base. In what follows, we present a brief reference of the ODeLP language. The interested reader can consult [3] for a more detailed version.

The language of ODeLP is based on the language of logic programming. Standard logic programming concepts (such as signature, variables, functions, etc) are defined in the usual way. Literals are atoms that may be preceded by the symbol "$\sim$" denoting *strict* negation, as in extended logic programming.

ODeLP programs are formed by *observations* and *defeasible rules*. Observations correspond to facts in the context of logic programming, and represent the knowledge an agent has about the world. *Defeasible rules* provide a way of performing tentative reasoning as in other argumentation formalisms [7].

Definition 3.1 An *observation* is a ground literal L representing some fact about the world, obtained through the perception mechanism, that the agent believes to be correct. A *defeasible rule* has the form $L_0 \prec L_1, L_2, \ldots, L_k$, where L_0 is a literal and $L_1, L_2, \ldots, L_k$ is a non-empty finite set of literals.

Definition 3.2 An *ODeLP program* is a pair $\langle \Psi, \Delta \rangle$, where Ψ is a finite set of observations and Δ is a finite set of defeasible rules. In a program $\mathcal{P}$, the set Ψ must be *non-contradictory* ($i.e.$, it is not the case that $Q \in \Psi$ and ~$Q \in \Psi$, for any literal Q).

[1]see [9] for a precise definition.

```
lion(simba).
lion(mufasa).
puppy(simba).
feline(X) -< lion(X).
climbs_tree(X) -< feline(X).
~climbs_tree(X) -< lion(X).
climbs_tree(X) -< lion(X), puppy(X).
~climbs_tree(X) -< sick(X).
```

Figure 1: An ODeLP program modeling the behavior of a group of lions

Example 3.1 Fig. 1 shows an ODeLP program for modeling the behavior of a group of lions. Observations describe that Mufasa is a lion, and Simba is a puppy lion. The rules establish that felines usually climb trees, lions usually don't. Exceptionally, puppy lions can climb trees. The remaining rule states that seriously sick animals cannot climb trees.

Given an ODeLP program $\mathcal{P}$, a query posed to $\mathcal{P}$ corresponds to a ground literal Q which must be supported by an *argument* [12]. Arguments are built on the basis of a *defeasible derivation* computed by backward chaining applying the usual SLD inference procedure used in logic programming. Observations play the role of facts and defeasible rules function as inference rules. In addition to provide a proof supporting a ground literal, such a proof must be non-contradictory and minimal for being considered as an argument in ODeLP. Formally:

Definition 3.3 Given a ODeLP program $\mathcal{P}$, an *argument* $\mathcal{A}$ for a ground literal Q, also denoted $\langle \mathcal{A}, Q \rangle$, is a subset of ground instances of the defeasible rules in $\mathcal{P}$ such that: (1) there exists a defeasible derivation for Q from $\Psi \cup \mathcal{A}$, (2) $\Psi \cup \mathcal{A}$ is non-contradictory, and (3) $\mathcal{A}$ is minimal with respect to set inclusion in satisfying (1) and (2).

Given two arguments $\langle \mathcal{A}_1, Q_1 \rangle$ and $\langle \mathcal{A}_2, Q_2 \rangle$, we will say that $\langle \mathcal{A}_1, Q_1 \rangle$ is a *sub-argument* of $\langle \mathcal{A}_2, Q_2 \rangle$ iff $\mathcal{A}_1 \subseteq \mathcal{A}_2$.

To use defeasible rules in arguments we must first obtain their *ground instances*, changing variables for ground terms, so that variables with the same name are replaced for the same term.

As in most argumentation frameworks, arguments in ODeLP can attack each other. This situation is captured by the notion of *counterargument*.

Definition 3.4 An argument $\langle \mathcal{A}_1, Q_1 \rangle$ *counter-argues* an argument $\langle \mathcal{A}_2, Q_2 \rangle$ at a literal Q if and only if there is a sub-argument $\langle \mathcal{A}, Q \rangle$ of $\langle \mathcal{A}_2, Q_2 \rangle$ such that Q_1 and Q are complementary literals.

Defeat among arguments is defined combining the counterargument relation and a preference criterion "$\preceq$". An argument $\langle \mathcal{A}_1, Q_1 \rangle$ *defeats* $\langle \mathcal{A}_2, Q_2 \rangle$ if $\langle \mathcal{A}_1, Q_1 \rangle$ is a counterargument of $\langle \mathcal{A}_2, Q_2 \rangle$ at a literal Q and $\langle \mathcal{A}_1, Q_1 \rangle \preceq \langle \mathcal{A}, Q \rangle$ (proper defeater) or $\langle \mathcal{A}_1, Q_1 \rangle$ is unrelated to $\langle \mathcal{A}, Q \rangle$ (*blocking defeater*).

Defeaters are arguments and may in turn be defeated. Thus, a complete dialectical analysis is required to determine which arguments are ultimately accepted. Such analysis results in a tree structure called *dialectical tree*, in which arguments are nodes labeled as undefeated (`U-nodes`) or defeated (`D-nodes`) according to a marking procedure. Formally:

Definition 3.5 The *dialectical tree* for an argument $\langle \mathcal{A}, Q \rangle$, denoted $\mathcal{T}_{\langle \mathcal{A}, Q \rangle}$, is recursively defined as follows:

1. A single node labeled with an argument $\langle \mathcal{A}, Q \rangle$ with no defeaters (proper or blocking) is by itself the dialectical tree for $\langle \mathcal{A}, Q \rangle$.

2. Let $\langle \mathcal{A}_1, Q_1 \rangle, \langle \mathcal{A}_2, Q_2 \rangle, \ldots, \langle \mathcal{A}_n, Q_n \rangle$ be all the defeaters (proper or blocking) for $\langle \mathcal{A}, Q \rangle$. The dialectical tree for $\langle \mathcal{A}, Q \rangle$, $\mathcal{T}_{\langle \mathcal{A}, Q \rangle}$, is obtained by labeling the root node with $\langle \mathcal{A}, Q \rangle$, and making this node the parent of the root nodes for the dialectical trees of $\langle \mathcal{A}_1, Q_1 \rangle, \langle \mathcal{A}_2, Q_2 \rangle, \ldots, \langle \mathcal{A}_n, Q_n \rangle$.

For the marking procedure we start labeling the leaves as `U-nodes`. Then, for any inner node $\langle \mathcal{A}_2, Q_2 \rangle$, it will be marked as `U-node` iff every child of $\langle \mathcal{A}_2, Q_2 \rangle$ is marked as a `D-node`. If $\langle \mathcal{A}_2, Q_2 \rangle$ has at least one child marked as `U-node` then it is marked as a `D-node`.

Dialectical analysis may in some situations give rise to *fallacious argumentation* [12]. In ODeLP, dialectical trees avoid fallacies applying additional constraints when building *argumentation lines* (the different possible paths in a dialectical tree). These constrains also avoid circular argumentation. The resulting kind of trees is called *Acceptable dialectical trees*. The interested reader can consult [12] where these issues are analyzed in detail in the context of the DeLP system.

Finally, the notion of warrant is grounded on acceptable dialectical trees. Given a query Q and an ODeLP program $\mathcal{P}$, we will say that Q is *warranted* wrt $\mathcal{P}$ iff there exists an argument $\mathcal{T}_{\langle \mathcal{A}, Q \rangle}$ such that the root of its associated dialectical tree $\mathcal{T}_{\langle \mathcal{A}, Q \rangle}$ is marked as a U-node.

Solving a query Q in ODeLP accounts for trying to find a warrant for Q, as shown in the following example.

Example 3.2 Consider the program shown in Example 3.1, and let `climbs_tree(simba)` be a query wrt that program. The search for a warrant for `climbs_tree(simba)` will result in an argument $\langle \mathcal{A},$ `climbs_tree(simba)`$\rangle$ with one defeater, $\langle \mathcal{B},$ ~`climbs_tree(simba)`$\rangle$ that is in turn defeated by $\langle \mathcal{C},$ `climbs_tree(simba)`$\rangle$. The structure of these arguments is detailed in Fig. 2.
Using specificity as the preference criterion, $\langle \mathcal{B},$ ~`climbs_tree(simba)`$\rangle$ is proper defeater for $\langle \mathcal{A},$ `climbs_tree(simba)`$\rangle$, but $\mathcal{B}$ is in turn properly defeated by $\langle \mathcal{C},$ `climbs_tree(simba)`$\rangle$. In this case `climbs_tree(simba)` is a warranted fact.
Suppose now we learn that Simba is sick. In ODeLP we can add this fact to the knowledge base using an updating function [3, 14]. Then, a new argument will arise that could not have been built before, $\langle \mathcal{D},$ ~`climbs_tree(simba)`$\rangle$ detailed Fig. 3.
Using specificity as the preference criterion, $\langle \mathcal{D},$ ~`climbs_tree(simba)`$\rangle$ is a blocking defeater for both $\langle \mathcal{A},$ `climbs_tree(simba)`$\rangle$ and $\langle \mathcal{C},$ `climbs_tree(simba)`$\rangle$. The resulting dialectical tree is shown Fig.3. Now, the marking procedure determines that the root node is a `D-node` and therefore `climbs_tree(simba)` is no longer warranted.

4 PRECOMPILED KNOWLEDGE IN ODELP

The ODeLP language was specifically designed to be integrated in practical applications. Therefore, the inference engine should be able to address real-time constrains that arise in these scenarios. To do this, we use precompiled knowledge to avoid recomputing arguments which were already computed before, in a TMS fashion.

The notion of *dialectical databases* is fundamental for precompiled knowledge in ODeLP. A dialectical database for a given program $\mathcal{P}$ collects a set of schematic arguments, called *potential arguments*, and the defeat relation among them. Every potential argument represents a set of arguments that are obtained using *different* instances of the *same* defeasible rules. This avoids generating and

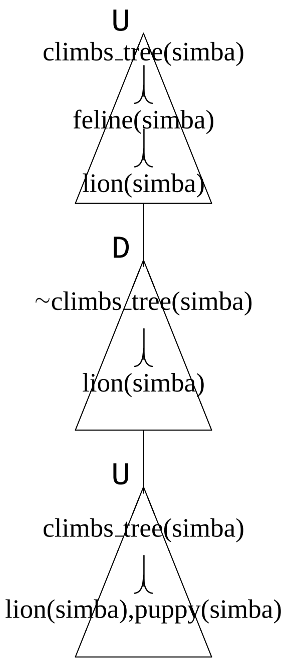

Figure 2: Dialectical tree from Example 3.2

storing many arguments which are structurally identical, only differing in the constant names being used to build the corresponding derivations. The dialectical database is also defined independently from the observation set Ψ, so it does not have to be changed if the set of observations is updated with new perceptions. Next we introduce a set of auxiliary notions that will be used to formally define dialectical databases.

Definition 4.1 Let A be a set of defeasible rules. A set $\mathcal{B}$ formed by ground instances of the defeasible rules in A is an *instance of* A iff every instance of a defeasible rule in $\mathcal{B}$ is an instance of a defeasible rule in A.

Example 4.1 If A ={ `s(X)` $\prec$ `~r(X)`; `~r(X)` $\prec$ `p(X)`} then $\mathcal{B}$ = { `s(t)` $\prec$ `~r(t)`; `~r(a)` $\prec$ `p(a)`} is an instance of A.

Definition 4.2 Let Δ be a set of defeasible rules. A subset A of Δ is a *potential argument* for a literal Q, noted as $\langle\!\langle \mathsf{A}, \mathsf{Q} \rangle\!\rangle$, if there exists a non-contradictory set of literals Φ and an instance $\mathcal{B}$ of the rules in A such that $\langle \mathcal{B}, Q \rangle$ is an argument wrt $\langle \Phi, \Delta \rangle$.

In the definition above the set Φ stands for a state of the world (set of observations) in which we can obtain the instance $\mathcal{B}$ from the set A of defeasible rules such that $\langle \mathcal{B}, Q \rangle$ is an argument (as stated in Def.3.3). Note that the set Φ must necessarily be non-contradictory to model a coherent scenario.

Precompiled knowledge associated with an ODeLP program $\mathcal{P} = \langle \Psi, \Delta \rangle$ will involve the set of all potential arguments that can be built from $\mathcal{P}$ as well as the defeat relation among them. Then, instead of computing a query for a given ground literal Q, the ODeLP interpreter will search for a potential argument A for Q such that a particular instance $\mathcal{B}$ of A is an argument for Q wrt $\mathcal{P}$.

To speed-up inference, the defeat relations among potential arguments must also be recorded, as we will see later on. To do this, we extend the concepts of counterargument and defeat for potential arguments. A potential argument $\langle\!\langle \mathsf{A}_1, \mathsf{Q}_1 \rangle\!\rangle$ *counter-argues* $\langle\!\langle \mathsf{A}_2, \mathsf{Q}_2 \rangle\!\rangle$ at a literal Q if and only if there is a potential sub-argument $\langle\!\langle \mathsf{A}, \mathsf{Q} \rangle\!\rangle$ of $\langle\!\langle \mathsf{A}_2, \mathsf{Q}_2 \rangle\!\rangle$ such that Q_1 and Q are contradictory literals.[2]

[2]Note that $P(X)$ and $\sim P(X)$ are contradictory literals although they are non-grounded. The same idea is applied to identify contradiction in potential arguments.

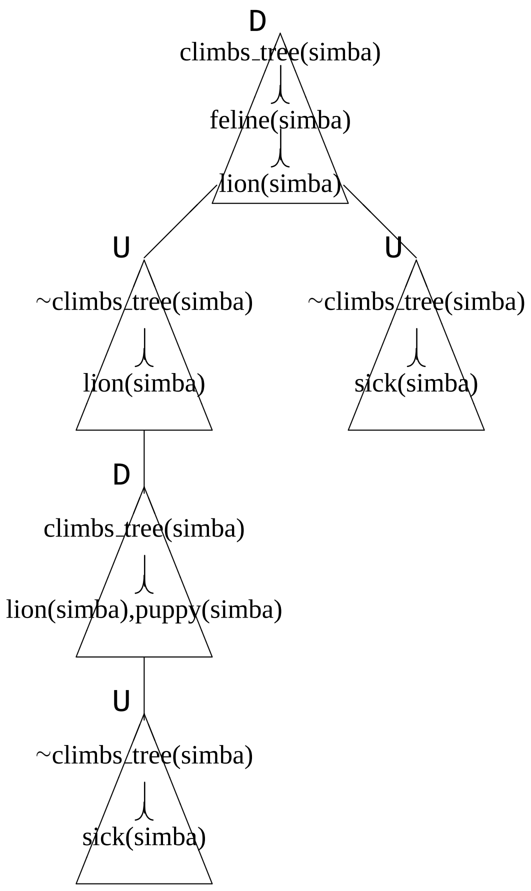

Figure 3: Final dialectical tree from Example 3.2

Note that potential counter-arguments may or may not result in a real conflict between the instances (arguments) associated with the corresponding potential arguments. In some cases instances of these arguments cannot co-exist in any scenario ($e.g.$, consider two potential arguments based on contradictory observations). The notion of defeat is also extended to potential arguments, redefining the preference criterion accordingly.

Finally, using potential arguments and their associated defeat relation, we can formally define the notion of *dialectical databases* associated with a given ODeLP program $\mathcal{P}$.

Definition 4.3 Let $\mathcal{P} = \langle \Psi, \Delta \rangle$ be an ODeLP program. The *dialectical database* of $\mathcal{P}$, denoted as $\mathcal{DB}_\Delta$, is a 3-tuple $(\mathit{PotArg}(\Delta), D_p, D_b)$ such that:

1. $\mathit{PotArg}(\Delta)$ is the set $\{\langle\langle \mathsf{A}_1, \mathsf{Q}_1 \rangle\rangle, \ldots, \langle\langle \mathsf{A}_k, \mathsf{Q}_k \rangle\rangle\}$ of all the potential arguments that can be built from Δ.

2. D_p and D_b are relations over the elements of $\mathit{PotArg}(\Delta)$ such that for every $(\langle\langle \mathsf{A}_1, \mathsf{Q}_1 \rangle\rangle, \langle\langle \mathsf{A}_2, \mathsf{Q}_2 \rangle\rangle)$ in D_p (respectively D_b) it holds that $\langle\langle \mathsf{A}_2, \mathsf{Q}_2 \rangle\rangle$ is a proper (respectively blocking) defeater of $\langle\langle \mathsf{A}_1, \mathsf{Q}_1 \rangle\rangle$.

Example 4.2 Consider the program in example 3.1. The dialectical database of $\mathcal{P}$ is composed by the following potential arguments:

- $\langle\langle \mathsf{A}_1, \texttt{climbs\_tree(X)} \rangle\rangle$,
  $\mathsf{A}_1 = \{\texttt{climbs\_tree(X)} \prec \texttt{feline(X)}\}$.

- $\langle\langle \mathsf{A}_2, \texttt{climbs\_tree(X)} \rangle\rangle$,
  $\mathsf{A}_2 = \{\texttt{climbs\_tree(X)} \prec \texttt{feline(X)}, \texttt{feline(X)} \prec \texttt{lion(X)}\}$.

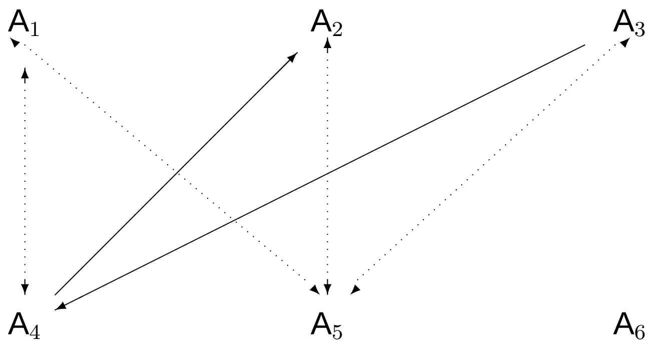

Figure 4: Dialectical database corresponding to Example 4.2.

- $\langle\langle \mathsf{A}_3, \texttt{climbs\_tree(X)}\rangle\rangle$,
  $\mathsf{A}_3 = \{\texttt{climbs\_tree(X)} \prec \texttt{lion(X), puppy(X)}\}$.
- $\langle\langle \mathsf{A}_4, \sim\texttt{climbs\_tree(X)}\rangle\rangle$,
  $\mathsf{A}_4 = \{\sim\texttt{climbs\_tree(X)} \prec \texttt{lion(X)}\}$.
- $\langle\langle \mathsf{A}_5, \sim\texttt{climbs\_tree(X)}\rangle\rangle$,
  $\mathsf{A}_5 = \{\sim\texttt{climbs\_tree(X)} \prec \texttt{sick(X)}\}$.
- $\langle\langle \mathsf{A}_6, \texttt{feline(X)}\rangle\rangle$,
  $\mathsf{A}_6 = \{\texttt{feline(X)} \prec \texttt{lion(X)}\}$.

and the defeat relations:

- $D_p = \{(\mathsf{A}_2, \mathsf{A}_4), (\mathsf{A}_4, \mathsf{A}_3)\}$
- $D_b = \{(\mathsf{A}_1, \mathsf{A}_4), (\mathsf{A}_4, \mathsf{A}_1), (\mathsf{A}_1, \mathsf{A}_5), (\mathsf{A}_5, \mathsf{A}_1), (\mathsf{A}_2, \mathsf{A}_5), (\mathsf{A}_5, \mathsf{A}_2), (\mathsf{A}_3, \mathsf{A}_5), (\mathsf{A}_5, \mathsf{A}_3)\}$.

The relations are also depicted in figure 4, where proper defeat is indicated with a normal arrow and blocking defeat is distinguished with a dotted arrow.

5 ALGORITHMS FOR BUILDING DIALECTICAL DATABASES

Given an ODeLP program $\mathcal{P}$, its dialectical database $\mathcal{DB}_\Delta$ can be understood as a graph from which *all* possible dialectical trees computable from $\mathcal{P}$ can be obtained. In previous work [3], it was already addressed how to use precompiled knowledge for computing warrants with respect to a given program. In this section we address how to build this graph for a given set of defeasible rules Δ.

To build the dialectical database for a given program we need to obtain every potential argument and record the defeat relation among them. This is done by algorithm BuildDialecticalDatabase. Briefly speaking, it first uses the algorithm ObtainPotentialArgs to select a of candidates that may be potential arguments for the set Δ in the set Candidates. Every member of this set is later analyzed to verify if it complies with the conditions present in definition 4.2. To do that, CreateInstance consistently replaces variables in a given potential argument for a set of literals. Then the argument obtained in $\langle \mathcal{A}, Q_1\rangle$ must be consistent and minimal (requirements present in definition 4.2) to be finally added in the set of PotArgs.

If the answer is positive, then it is selected as a potential arguments and its defeaters are found using the algorithm FindDefeaters that compares the potential argument to be added into the set with the potential arguments already considered to update the defeat relations D_b and D_p.

Algorithm 1 BuildDialecticalDatabase

```
input:  $\Delta$
output:  PotArgs,$D_p$,$D_b$ //(a dialectical database)

PotArgs := $\emptyset$
ObtainPotentialArgs($\Delta$, Candidates)
//Finds the set of potential arguments
  For every $\langle\langle$A,Q$\rangle\rangle$ in Candidates
    CreateInstance($\langle\langle$A,Q$\rangle\rangle$, $\langle \mathcal{A}, Q_1 \rangle$)
    $\Psi$ := $G(\langle \mathcal{A}, Q_1 \rangle)$
    //Calculates the ground for $\mathcal{A}$, that is the literals
    //in $\mathcal{A}$ that do not appear in the head of a rule
    If Literals($\langle \mathcal{A}, Q_1 \rangle$) is not contradictory and
    $G(\mathcal{A}) \cup \mathcal{A} \mathrel{|\!\sim} Q_1$ and no $\mathcal{A}' \subset \mathcal{A}$ is such that $G(\mathcal{A}') \cup \mathcal{A}' \mathrel{|\!\sim} Q_1$ then
      FindDefeaters(PotArgs, $\langle\langle$A,Q$\rangle\rangle$, $D_p$, $D_b$)
      PotArgs := PotArgs $\cup\{\langle\langle$A,Q$\rangle\rangle\}$
```

Next, we analyze the auxiliary algorithms used by BuildDialecticalDatabase. The algorithm ObtainPotentialArgs finds the set of potential arguments using backward chaining from every rule in Δ. This is an smart way to build this set, that results in computational gains with respect to finding all the set of rules that can be obtained from Δ. First, it chooses a rule to guide the backward chaining. Then, it uses the algorithm FindCandidates that recursively considers every potential argument that can be found starting with that rule. This algorithm also marks rules that have been already used to avoid re-computing potential arguments that have been already added into the set of candidates.

Finally, CreateInstance consistently replaces variables in a given potential argument for a set of literals. It uses backward chaining and composes substitutions to build the instance, if any exists. This algorithm requires defeasible rules in the set A to be standarized apart so that they do not contain common variables. That is, for any pair of rules r_1, r_2 in A it must hold that the intersection between the set of variables in r_1 and the set of variables in r_2 is empty.

Algorithm 2 Obtain Potential Arguments

```
input:  $\Delta$
output:   Candidates

Candidates := $\emptyset$
Marked := $\emptyset$
  For every rule such that $r \in \Delta$ and $r \notin$ Marked
    FindCandidates(r, NewCandidates)
    Candidates := Candidates $\cup$ NewCandidates
```

Algorithm 3 FindCandidates

input: $r = \alpha \prec \beta$ *//uninstanciated rule*
output: Cand *//set of candidates found from r*

```
Cand := $\{\langle\!\langle\{\alpha \prec \beta\}, \alpha\rangle\!\rangle\}$
For every literal $p \in \beta$ such that
there is a rule with $p$ in the head, $p \prec \gamma$
    FindCandidates($p \prec \gamma$, $C$)
    For every $C_i \subset C$, $C_i <> \emptyset$
      Cand := Cand $\cup\{\langle\!\langle\{\alpha \prec \beta\} \cup C_i, \alpha\rangle\!\rangle\}$
    Marked := Marked $\cup\{\alpha \prec \beta\}$
```

Algorithm 4 CreateInstance

input: $\langle\!\langle$A, Q$\rangle\!\rangle$ *//a possible potential argument*
output: $\langle \mathcal{A}, Q_1\rangle$ *//an argument built from the rules in $\langle\!\langle$A, Q$\rangle\!\rangle$, if any*

```
CreateStack(S)
Instanciate(Q, $Q_1$) //Sets as goal an instance of Q
push($Q_1$,S)
$\theta$:= {}
While S is not empty
  goal := pop(S)
  If there exists a rule $r$ in A and a substitution $\sigma$
    such that head($r$)$\sigma$ = goal
  then
    new_body := apply $\sigma$ to the body of the rule $r$
    $\theta$ := compose $\theta$ and $\sigma$
    push(new_body,S)
  else
    $r$ := pop(S)
    If there exists a substitution $\sigma$ and an observation $\alpha$
      such that $r\sigma$ = $\alpha$
    then $\theta$ := compose $\theta$ and $\sigma$
    else fail {It is not possible to find an instance}
$\mathcal{A}$ := apply $\theta$ to every rule in A
Return($\langle \mathcal{A}, Q_1\rangle$)
```

Figure 5: Algorithm to obtain an instance of a potential argument.

5.1 Complexity results

In this section we analyze the complexity of algorithm BuildDialecticalDatabase since this algorithm resumes the construction process of ODeLP's precompiled knowledge.

To do this, we first consider the complexity of auxiliary algorithms. Note that the analysis presented here holds for ODeLP programs with a finite Herbrand base. We plan to extend this analysis in future work to full ODeLP programs.

CreateInstance consistently replaces variables in a given potential argument for a set of literals. This task is analogous to the following decision problem: *is a given subset of defeasible rules an argument for a literal from a given program P?.* In [4] this is shown to be a **P**-complete problem for the DeLP system. This result is an upper bound for ODeLP, where there is no strict knowledge and thus complexity is clearly reduced.

ObtainPotentialArguments returns every set of rules that may be a potential argument for Δ. A rough upper bound for the number of potential arguments is $2^{|\Delta|}$. Therefore, Obtain potential arguments is in $O(2^{|\Delta|})$.

Algorithm FindDefeaters must compare the potential argument to be added with every potential argument that is already in the set `PotArgs`. This is also in $O(2^{|\Delta|})$.

Finally, we analyze algorithm BuildDialecticalDatabase. It first calls ObtainPotentialArguments. Then, for every argument in the set `Candidates`, it does following four tasks:

1. Calls algorithm CreateInstance.

2. Checks consistency: this check depends on the number of literals in the argument, that can be bounded by the number of literals in the signature of the program, noted by $|Lit|$. Thus, this task is in P.

3. Checks minimality: a simple algorithm for verifying whether a set of defeasible rules is minimal with respect to set inclusion (for entailing a given literal l) would delete every rule at a time and verify if the remaining set of rules can entail l. Worst case of the minimality condition is considered when we assume that the argument has $|\Delta|$ defeasible rules. In this case computing minimality condition takes $|\Delta|$ to verify that l cannot be entailed for a subset of the rules in the potential argument. Then every loop is in P and the problem of checking minimality is solvable in polynomial time.

4. Calls algorithm FindDefeaters.

Therefore the cost of the loop is in $O(2^{|\Delta|})$ and the number of times it is executed is bounded by $2^{|\Delta|}$. Then algorithm BuildDialecticalDatabase is in Σ_p^2, that is, the second level of the polynomial hierarchy.[3]

6 CONCLUSIONS AND FUTURE WORK

The notion of dialectical databases was proposed in [3] to comply with real time requirements needed to model agent reasoning in dynamic environments. In this paper we have devised a set of algorithms for the construction of the precompiled knowledge component in ODeLP.

We have also analyzed the complexity of these algorithms from a theoretical standpoint. Even though the algorithms are computationally expensive we must recall that the task of building precompiled knowledge is performed only once, after codifying the program. Moreover, the dialectical database is not affected by changes in the program's observations and the set of rules is not expected to change in applications using ODeLP.

As future work, we will analyze how the use of precompiled knowledge in the inference process reduces complexity in ODeLP. We also plan to extend the complexity analysis, currently valid for programs with a finite Herbrand base, to full ODeLP programs.

[3]The interested reader may consult [1] for more information on the polynomial hierarchy.

REFERENCES

[1] D.P. Bovet and P. Crescenzi. *Introduction to the theory of Complexity*. Prentice Hall International, 1994.

[2] A. L. Brown. Modal Propositional Semantics for Reason Maintenance Systems. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 178–183, June 1985.

[3] M. Capobianco, C. Chesñevar, and G. Simari. Argumentation and the dynamics of warranted beliefs in changing environments. *Journal of Autonomous Agents and Multiagent Systems*, 11:127–151, 2005.

[4] L. Cecchi, P. Fillottrani, and G. Simari. On the complexity of delp through game semantics. In *Proc. 11th Intl. Workshop on Nonmonotonic Reasoning (NMR 2006)*, pages 386–394, 2006.

[5] C. Chesñevar and A. Maguitman. An argumentative approach to assesing natural language usage based on the web corpus. In *Proc. of European Conference on Artificial Intelligence (ECAI 2004). Valencia, Spain*. ECCAI, August 2004.

[6] C. Chesñevar and A. Maguitman. ARGUENET: An Argument-Based Recommender System for Solving Web Search Queries. In *Proc. of Intl. IEEE Conference on Intelligent Systems IS-2004. Varna, Bulgaria*, June 2004.

[7] C. Chesñevar, A. Maguitman, and R. Loui. Logical Models of Argument. *ACM Computing Surveys*, 32(4):337–383, 2000.

[8] J. de Kleer. A comparison of ATMS and CSP techniques. In N. S. Sridharan, editor, *Proceedings of the 11th International Joint Conference on Artificial Intelligence, Workshop on Practical Reasoning and Rationality*, pages 290–296, Detroit, USA, August 1989. Morgan Kaufmann.

[9] Jon Doyle. A Truth Maintenance System. *Artificial Intelligence*, 12(3):231–272, 1979.

[10] Charles Elkan. A Rational Reconstruction of Nonmonotonic Truth Maintenance Systems. *Artificial Intelligence*, 43:219–234, 1990.

[11] K. Forbus and J. de Kleer. *Building Problem Solvers*. MIT Press, Cambridge, Massachusetts, 1993.

[12] A. García and G. Simari. Defeasible Logic Programming: An Argumentative Approach. *Theory and Practice of Logic Programming*, 4(1):95–138, 2004.

[13] S. Gomez and C. Chesñevar. A Hybrid Approach to Pattern Classification Using Neural Networks and Defeasible Argumentation. In *Proc. of Intl. 17th FLAIRS Conference. Palm Beach, FL, USA*, pages 393–398. AAAI, May 2004.

[14] Hirofumi Katsuno and Alberto Mendelzon. On the difference between updating a knowledge base and revising it. In P.Gardenfors, editor, *Belief Revision*, pages 183–203. Cambridge University Press, 1992.

[15] D. McAllester. Truth Maintenance. In Reid Smith and Tom Mitchell, editors, *Proceedings of the 8th National Conference on Artificial Intelligence*, volume 2, pages 1109–1116. American Association for Artificial Intelligence, AAAI Press, August 1990.

[16] H. Prakken and G. Vreeswijk. Logical systems for defeasible argumentation. In *Handbook of Philosophical Logic*, volume 4, pages 219–318. 2002.

[17] G. R. Simari and R. P. Loui. A Mathematical Treatment of Defeasible Reasoning and its Implementation. *Artificial Intelligence*, 53(1–2):125–157, 1992.