From Proof Trees to Justificatory Answering

Newton José Vieira, Isabel Gomes Barbosa, and Carlos Camarão de Figueiredo

Universidade Federal de Minas Gerais,
Belo Horizonte, Minas Gerais, Brazil.
{nvieira,isabel,camarao}@dcc.ufmg.br

Abstract. In those situations in which a question asks for information about an individual, the answer may depend on general domain knowledge, but usually it also depends on specific characteristics of that individual. If the specific characteristics are incomplete or even absent from the knowledge base, the general knowledge can point to several alternative answers. In order to decide which of them applies to that particular individual it is necessary to complete the lacking knowledge. In this paper we present an approach to this problem in the context of a first-order knowledge base. The general problem here is that a question of the form *find* X *such that* $q(X)$, where q is a predicate symbol and X is an n-tuple of variables, may have in the usual proof systems an answer $q(T_1) \vee \ldots \vee q(T_k)$, $k \geq 2$, where each T_i is an n-tuple of terms; and in some scenarios such a disjunctive answer may be considered too imprecise to be helpfull. Here we present a method that allows, from a proof tree resulting from a deduction of $q(T_1) \vee \ldots \vee q(T_k)$, the contruction of a *justificatory answer*, an answer that gives conditions under which each disjunct $q(T_i)$ is true. Each condition of a justificatory answer gives a "concise" justification of a disjunct resembling an abductive explanation for the disjunct, but a special one in that each condition of a justificatory answer is derived from an actual proof of $q(T_1) \vee \ldots \vee q(T_k)$. As in abductive explanation, in general several alternative justificatory answers are possible, and the choice of a "good" one depends on knowledge of the appropriate vocabulary.

1 Introduction

Given a knowledge base expressed in a first-order language, a question of the form *find* X *such that* $q(X)$, where q is a predicate symbol and X is an n-tuple of variables, may have in the usual proof systems an answer $q(T_1) \vee \ldots \vee q(T_k)$, where each T_i is an n-tuple of terms [7]. If $k = 1$, the answer is called a *categorical* answer, and if $k \geq 2$, it is termed a *disjunctive* answer. A disjunctive answer is often considered less informative than a categorical answer, since it doesn't specify which of its disjuncts is actually a correct answer to the question. Indeed, when a disjunctive answer is obtained to a non-disjunctive question, it usually means that some specific information required to deduce a categorical answer are missing from the knowledge base. If such information could be retrieved and made available to the user, then he might be able to get a categorical answer

to his question. Assuming a disjunctive answer of the form $q(T_1) \vee \ldots \vee q(T_k)$, $k \geq 2$, one way of doing this is to find and exhibit conditions under which each disjunct $q(T_i)$, $1 \leq i \leq k$, is true. An incipient manner of finding such conditions in the context of resolution proof procedures was introduced in [5].

As an example, adapted from [5], consider the (very simple) knowledge base defined by the clauses:

1. $\neg$adult(x) $\vee$ prescribe(x,A)
   (*If someone is an adult, prescribe drug A*)
2. adult(x) $\vee$ prescribe(x,B)
   (*If someone is not an adult, prescribe drug B*)

and the question

find x such that prescribe(John,x)
(*what drug should be prescribed to John?*)

with logical form "$\exists x$ prescribe(John,x)". Most proof systems would give the answer prescribe(John,A) $\vee$ prescribe(John,B). From this answer, one cannot tell which drug, A or B, should be prescribed to John. However, if the answer were

adult(John) $\rightarrow$ prescribe(John,A)
$\neg$adult(John) $\rightarrow$ prescribe(John,B)

and the user knew whether John is an adult or not, he would know which drug to prescribe. This kind of answer is called a *justificatory answer*, as each condition in an antecedent "justifies" the disjunct in the consequent.

The above example illustrates the general situation in which a question asks for information about an individual and such information depends, not only on general knowledge of the domain, but also on specific characteristics of that individual. If the specific characteristics are incomplete or even absent from the knowledge base, the general knowledge can point to several alternative answers. In order to decide which of them applies to that particular individual it is necessary to complete the lacking knowledge (in the example, the knowledge whether John is an adult or not).

The main purpose of this paper is to present a method that allows one to take a proof tree derived from a knowledge base Σ and a question $q(X)$, to which corresponds a disjunctive answer of the form $q(T_1) \vee \ldots \vee q(T_k)$, $k \geq 2$, and produce a justificatory answer of the form:

$$\begin{array}{l} \alpha_1 \rightarrow q(T_1) \\ \alpha_2 \rightarrow q(T_2) \\ \vdots \\ \alpha_k \rightarrow q(T_k) \end{array}$$

where $\Sigma \models \alpha_i \rightarrow q(T_i)$, for $1 \leq i \leq k$. Moreover, it is assumed that each T_i is an n-tuple of ground terms, and so the original answer is a "specific answer" according to the classification of Burhans and Shapiro [4].

Regretably, it is not always possible to have an answer of the above format, in particular if the knowledge is intrinsically disjunctive. For example, if you ask the question *what subject does John teach*, and the knowledge base contains the fact that *John teaches Mathematics or Logic*, then from this fact alone it follows immediately the statement *John teaches some subject*. A corresponding proof tree does not contain information that allows the extraction of one (non trivial) condition that implies *John teaches Mathematics* and another that implies *John teaches Logic*.

On the other hand, there can be several justificatory answers obtainable from a proof tree. Each one is extracted from a different *answer proof graph*. The method consists of two stages: first, it shows how to construct an answer proof graph from a proof tree; second, it specifies how to extract the justificatory answer from an answer proof graph. An initial work in the direction of presenting justificatory answers to a question is shown in [1].

As far as we know, the use of justificatory answers for the clarification of disjunctive answers constitutes original work.

This paper is organized as follows. Section 2 presents a definition of proof tree, shows how information is extracted from a proof tree and introduces important concepts to be used in next sections. Section 3 presents proof graphs, a generalization of proof trees. A definition of answer proof graphs and a method for constructing a justificatory answer from an answer proof graph is described in Section 4. Finally, conclusions are presented in Section 5.

2 Proof Trees

Proof trees are essentially what in [8] are defined as clausal tableaux and in [13] are referred to as proof trees (this last one, if suitably extended for all first order logic). They are also what Loveland calls goal trees in his MESON system [11][10]. Despite having a definition not entirely identical to the definitions given in the cited works, proof trees as defined here carry the same information: the actual unsatisfiable set of instances of clauses structured in the form of a tree.

From now on the terminology usually associated to trees will be used without explicit definitions. These definitions can be found in standard texts on discrete mathematics or graph theory. In particular, the concepts of root, leaf, father, child, ancestor, descendant, and so on, are used frequently ahead. In this paper ancestors and descendants of a node v will *not* include v itself.

Each node of a proof tree is labeled by a literal. In the sequel a subtree that has a literal L as its label is denoted by LS, where S denotes the (possibly empty) set of its children. A proof tree is constructed from a set of clauses through application of three rules. The first rule, the *codification* rule, is used only to initiate the construction of a proof tree. It corresponds to constructing an initial chain in model elimination [9]. The other two rules, *expansion* and *reduction*, are similar to the extension and reduction rules of model elimination [9]. There is no rule corresponding to the contraction rule of model elimination, and this is essentially what keeps all information used in the proof of unsatisfiability

recoverable from the (final) proof tree. The structure provided by proof trees is similar to that proposed by the MESON system of Loveland [10]. In the process of constructing a proof tree, the set S of a subtree LS can be absent; in this case, L is called a *candidate literal*. In fact, the tree that has such a subtree is a *partial* proof tree. In the process of constructing a proof tree, a partial proof tree evolves until there is no subtree LS with S absent, in other words, until there is no candidate literal.

Two special literals will be used ahead, both *not present in the knowledge base*: $\bot$ (*falsum*), to be thought as always false, and $\top$ (*verum*), to be interpreted as true, as usual. The notation $\overline{L}$ is used to refer to the complement of literal L (L and $\overline{L}$ are said to have opposite signs), while $|L|$ is used to refer to the atom of literal L. It must be supposed that $\overline{\bot} = \top$ and $\overline{\top} = \bot$. The notation $LS\sigma$ denotes the tree obtained by applying the substitution σ to the whole tree LS. As will be clear ahead, there are four types of literals labeling nodes in a proof tree: $\bot$ (labels the root), $\top$ (labels a leaf), *expanded* literal (labels a node that is not the root neither a leaf) and *reduced* literal (labels a leaf).

The three rules follow.

(Codification rule) The codification of an input clause $L_1 \vee \ldots \vee L_n$ produces the partial proof tree $\bot\{\overline{L}_1, \ldots, \overline{L}_n\}$. Thus, the root of every proof tree is labeled $\bot$. Literals $\overline{L}_1, \ldots, \overline{L}_n$ labeling the n children are candidate literals.

(Expansion rule) The expansion of a partial proof tree $\bot\{\ldots L \ldots\}$, where L is a candidate literal, using an input clause $M_1 \vee \ldots \vee M_n$ such that L and M_1 are unifiable with a most general unifier (mgu) σ, produces the tree $\bot\{\ldots L\{\overline{M}_2, \ldots, \overline{M}_n\} \ldots\}\sigma$, if $n \geq 2$, or $\bot\{\ldots L\{\top\{\}\} \ldots\}\sigma$ if $n = 1$. The literal $L\sigma$ in the resulting tree is an *expanded literal* and $\overline{M}_2\sigma, \ldots, \overline{M}_n\sigma$, if any, are candidate literals. Each other literal $N\sigma$ inherits its type from the corresponding literal N of the previous partial proof tree.

(Reduction rule) The reduction of a proof tree $\bot\{\ldots L\{\ldots M \ldots\} \ldots\}$, where M is a candidate literal of opposite sign to L, and $|M|$ and $|L|$ are unifiable with mgu σ, produces the tree $\bot\{\ldots L\{\ldots M\{\} \ldots\} \ldots\}\sigma$. The literal $M\sigma$ in the resulting tree is a *reduced literal*. Every other literal $N\sigma$ inherits its type from the corresponding literal N of the previous partial proof tree.

As said above, every deduction starts with the application of the codification rule, which produces the initial partial proof tree. A partial proof tree without candidate literals is what is called a *proof tree*. The only occurrence of $\bot$ appears at its root and each leaf is labeled with a reduced literal or $\top$; all internal nodes except the root are labeled with expanded literals.

Example 1. Consider the set of clauses:

1. $P(x) \vee R(x) \vee Q(x, y)$
2. $\neg Q(x, y) \vee S(x)$
3. $\neg S(x) \vee \neg Q(x, b)$
4. $\neg R(a)$

and the question *find* x *such that* $P(x)$. As required, one first negates $\exists x\, P(x)$ to obtain the clause:

5. $\neg P(x)$

A proof tree generated from clauses 1-5 is represented graphically in Figure 1. Expanded literals and $\bot$ are shown inside normal rectangles and reduced literals and $\top$ inside dashed rectangles. In that figure, to each clause used to construct

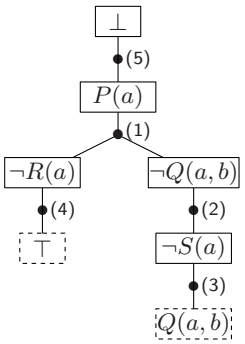


Fig. 1. A proof tree generated from clauses 1-5.

the tree via a rule is associated a node which links the nodes whose labels constitute the antecedent to the node labeled by the consequent of the corresponding clause instance (viewing an instance as a conditional in the usual manner).

Consistently with the discussion in Example 1, in the rest of this paper a proof tree will be seen as a bipartite graph with two kinds of nodes: *clause nodes*, whose labels are clauses from the knowledge base, and *literal nodes*, that have literals as its labels. The father of a clause node v will be called its *consequent node* and the children of v its *antecedent nodes*. The following definition introduces the concept of *support* of a clause node, which when properly generalized in the next section will be the basis for justificatory answers. A (clause or literal) node v labeled X will be refered to as the (clause or literal) node $v{:}X$.

Definition 1. *The* support *of a clause node* v, $S(v)$, *is defined as follows.*

- *If* v *has one antecedent node labeled* $\top$, $S(v) = \emptyset$;
- *If* v *has* n *antecedent nodes labeled with reduced literals* $L_1, \ldots, L_n$, $n \geq 1$, *and no more antecedent nodes,* $S(v) = \{L_1, \ldots, L_n\}$;
- *If* v *has* n *antecedent nodes labeled with reduced literals* $L_1, \ldots, L_n$, $n \geq 0$, *and it has* k *antecedent nodes labeled with expanded literals* $p_1 : M_1, \ldots, p_k : M_k$, $k \geq 1$, *then* $S(v) = \{L_1, \ldots, L_n\} \cup \bigcup_{1 \leq i \leq k} (S(u_i) - \{\overline{M_i}\})$, *where* $u_1, \ldots, u_k$ *are the clause nodes whose consequent nodes are* $p_1, \ldots, p_k$. □

Notice that a reduced literal is "propagated" into supports of ancestor clause nodes until its complement is found. As an example, the support of each clause node in Figure 1 is shown in Figure 2 at the node's left side.

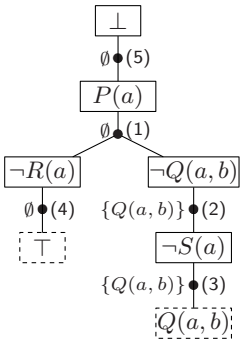


Fig. 2. Supports of clause nodes of proof tree of Figure 1.

Some important notation is introduced ahead in order to simplify the subsequent discussion.

Notation: Let S, R be sets of literals. Then:[1]

- $\overline{S} = \{\overline{L} \,|\, L \in S\}$;
- $\mathit{conj}(S) = \begin{cases} \top & \text{if } S = \emptyset, \\ M_1 \wedge \ldots \wedge M_n & \text{if } S = \{M_1, \ldots, M_n\}, n \geq 1; \end{cases}$
- $\mathit{disj}(S) = \begin{cases} \bot & \text{if } S = \emptyset, \\ M_1 \vee \ldots \vee M_n & \text{if } S = \{M_1, \ldots, M_n\}, n \geq 1; \end{cases}$
- $R \rightsquigarrow S = \mathit{conj}(R) \rightarrow \mathit{disj}(S)$. □

It is trivial do show that $R \rightsquigarrow S \equiv \mathit{disj}(\overline{R} \cup S)$.

Now, the support $S(v)$ of a clause node v and its consequent node $u : L$ are related, via the Theorem below, to a *lemma*, a usually simple formula that follows from the knowledge base and can be expressed as $S(v) \rightsquigarrow \{L\}$ or $\mathit{disj}(\overline{S(v)} \cup \{L\})$.

Theorem 1. *Let Σ be the set of clauses from which a proof tree T is constructed and v be a clause node of T that has a consequent node labeled L. Then $\Sigma \models \mathit{disj}(\overline{S(v)} \cup \{L\})$.*[2]

Next section presents the concept of *proof graph*, an acyclic connected graph that generalizes the concept of proof tree and is the starting point for obtaining justificatory answers. But before presenting proof graphs, it is shown next how an answer can be extracted from a proof tree.

In this paper it is supposed that a question is represented by an atomic formula. This assumption does not restrict generality. If the logical form of the question is a general formula $\exists X \mathcal{F}(X)$, where X is the n-tuple of variables that occur free in $\mathcal{F}(X)$, one can add the clausal form of $\forall X \ (\mathcal{F}(X) \rightarrow q(X))$ to

[1] $\mathit{conj}(S)$, $\mathit{disj}(S)$ and $R \rightsquigarrow S$ denote formulas. The last one denotes a conditional whose antecedent is $\mathit{conj}(S)$ and consequent is $\mathit{disj}(S)$.

[2] Theorem 1 follows immediately from Theorem 3, subsequently presented, since a proof tree is a special proof graph where each clause node has a single literal as its consequent.

the knowledge base, where q is a new predicate symbol, and the question is considered to be $q(X)$. This strategy is used in several works, such as [6][4].

A proof tree produced in order to process a question $q(X)$ has the following characteristics:

1. if there exists no node labeled $q(T)$ or $\neg q(T)$, where T is an instance of X, the knowledge base is inconsistent;
2. if there exists only one such node, then either (a) its label is positive and it is the only antecedent of a clause node whose consequent node is the root (labeled $\bot$), or (b) its label is negative and it is the consequent of a clause node whose antedecent is a leaf labeled $\top$;
3. otherwise, there are two possibilities: (a) the label of one of them is positive and it is the only antecedent of a clause node whose consequent is the root labeled $\bot$, and the others have labels that are all negative and each one of them is the consequent of a clause node whose antecedent is a leaf labeled $\top$, or (b) all of them have labels that are negative and each one of them is the consequent of a clause node whose antecedent is a leaf labeled $\top$.

In case 1, the answer, $\forall X q(X)$, follows from an inconsistent knowledge base. In case 2, the answer is categorical: $q(T)$, where T is an n-uple of terms. Finally, in case 3 the answer is disjunctive: $q(T_1) \vee \ldots \vee q(T_k)$, $k \geq 2$, where each T_i is an n-tuple of terms. Then, in general an answer extracted from a proof tree has the form $q(T_1) \vee \ldots \vee q(T_k)$ iff for each i $q(T_i)$ is the label of the antecedent of a clause node whose consequent is the root or $\neg q(T_i)$ labels the consequent of a clause node whose antecedent is a leaf labeled $\top$.

Example 2. Consider a knowledge base containing the following set of clauses:

1. $\mathsf{stud}(x) \vee \mathsf{staff}(x) \vee \mathsf{visit}(x)$
   (*Students, staff, visitors are eligible to borrow books*)
2. $\neg\mathsf{stud}(x) \vee \mathsf{under}(x) \vee \mathsf{grad}(x)$
   (*Students are divided into under and graduate students*)
3. $\neg\mathsf{stud}(x) \vee \neg\mathsf{under}(x) \vee \mathsf{borrow}(x,4)$
   (*Undergraduate students can borrow up to 4 books*)
4. $\neg\mathsf{stud}(x) \vee \neg\mathsf{grad}(x) \vee \mathsf{borrow}(x,8)$
   (*Graduate students can borrow up to 8 books*)
5. $\neg\mathsf{staff}(x) \vee \mathsf{acad}(x) \vee \mathsf{adm}(x)$
   (*Staff can be either academic or administrative*)
6. $\neg\mathsf{staff}(x) \vee \neg\mathsf{acad}(x) \vee \mathsf{borrow}(x,8)$
   (*Academic staff can borrow up to 8 books*)
7. $\neg\mathsf{staff}(x) \vee \neg\mathsf{adm}(x) \vee \mathsf{borrow}(x,2)$
   (*Administrative staff can borrow up to 2 books*)
8. $\neg\mathsf{visit}(x) \vee \mathsf{borrow}(x,4)$
   (*Visitors can borrow up to 4 books*)
9. $\neg\mathsf{under}(\mathsf{J})$
   (*John is not an undergraduate*)

Consider the question *how many books can John borrow*, formulated as *find y such that borrow(J,y)*. The clause obtained after negating the question is:

10. $\neg\mathsf{borrow}(\mathsf{J},y)$

A proof tree is given in Figure 3 along with the supports of all its clause nodes. Leting Σ be the set of clauses 1 to 9, Theorem 1 allow us to infer that

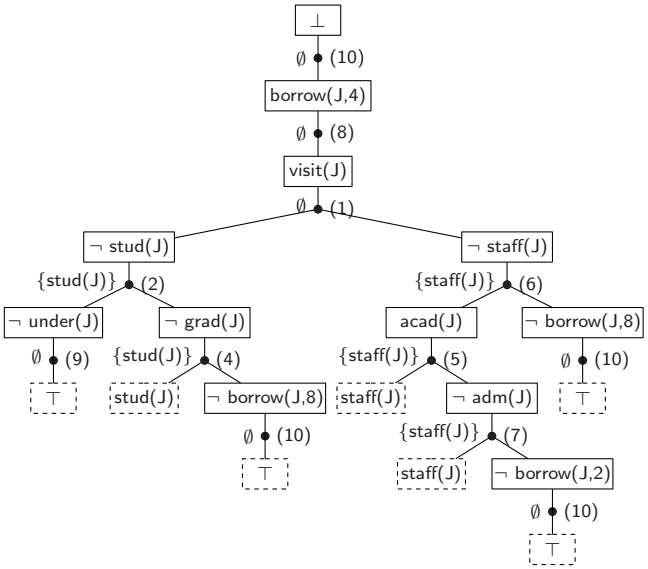


Fig. 3. Proof tree for Example 2.

$\Sigma \cup \{\neg\mathsf{borrow(J,2)}, \neg\ \mathsf{borrow(J,4)}, \neg\ \mathsf{borrow(J,8)}\} \models \mathit{disj}(\emptyset \cup \{\bot\})$, from which follows $\Sigma \models (\neg\ \mathsf{borrow(J,2)} \wedge \neg\ \mathsf{borrow(J,4)} \wedge \neg\ \mathsf{borrow(J,8)}) \rightarrow \bot$, thus justifying the disjunctive answer:

$$\mathsf{borrow(J,2)} \vee \mathsf{borrow(J,4)} \vee \mathsf{borrow(J,8)}.$$

3 From Proof Trees to Proof Graphs

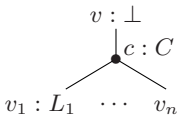
In a proof tree, a clause instance is represented as a conditional with an antecedent constituted of a conjunction of one or more literals and a consequent of a single literal. In a proof graph, a clause instance is also represented as a conditional, but with the consequent consisting of a *disjunction* of literals. As in a proof tree, each internal node is labeled with a literal of a clause instance and between the nodes labeled with the literals of the antecedent and the nodes labeled with the literals of the consequent there exists a node labeled with the original clause from the knowledge base. Again, the nodes labeled with literals are called *literal nodes* and those labeled with clauses are termed *clause nodes*.

A proof graph is a *bipartite* graph: a literal node labeled with a literal from an antecedent is linked to a clause node and this is linked to a literal node labeled with a literal from the consequent. As in a proof tree, an antecedent node is linked to a single clause node. In addition, a consequent node is a successor of a single clause node.

Starting from a proof tree as the initial proof graph, other proof graphs can be produced by applying the basic transformation step. The result will not necessarily be a tree, but will continue to be a directed acyclic connected (bipartite) graph. In order to explain some details of the basic transformation step, we define first the possible formats of a proof graph.

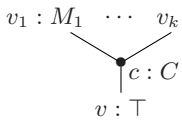
Definition 2. *A proof graph is an acyclic connected directed graph with clause and literal nodes in which each clause node c must be in a subgraph of one of the following forms, where* $v : L$ *means* node v labeled L*, and neither* L_i *nor* M_j *is* $\top$ *or* $\bot$ *(for any* i, j*):*

1.



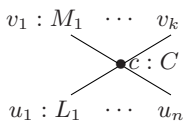
Node v has no successors (it is a root). Each v_i *must have exactly one clause node as predecessor and* L_i *is an expanded literal. Clause C has* $\overline{L_1} \vee \ldots \vee \overline{L_n}$ *as an instance.*

2.



Node v has no predecessors (it is a leaf). Each v_i *must have exactly one clause node as successor and* M_i *is an expanded literal. Clause C has* $M_1 \vee \ldots \vee M_k$ *as an instance.*

3.



Each v_i *must have exactly one clause node as successor and* M_i *is an expanded literal. Each* u_i *either has no predecessors (is a leaf), and in this case* L_i *is a reduced literal, or has exactly one clause node as predecessor and* L_i *is an expanded literal. Clause C has* $\overline{L_1} \vee \ldots \vee \overline{L_n} \vee M_1 \vee \ldots \vee M_k$ *as an instance.*

In a proof graph, each expanded literal $v : L$ is shared by instances of two clauses $c_1 : C_1$ and $c_2 : C_2$ such that (see the left side of Figure 4):

- node v is predecessor of c_1 and successor of c_2;
- the instance of C_1 can be written as $(\{L\} \cup \mathcal{B}) \rightsquigarrow \mathcal{A}$, where:
  - literals in $\mathcal{A}$ label successors of c_1 (if $\mathcal{A} = \emptyset$, there is only one successor and it is labeled $\bot$),
  - literals in $\mathcal{B}$ label predecessors of c_1 (no predecessors if $\mathcal{B} = \emptyset$);
- the instance of C_2 can be written as $\mathcal{D} \rightsquigarrow (\mathcal{C} \cup \{L\})$,
  - literals in $\mathcal{C}$ label successors of c_2 (no successors if $\mathcal{C} = \emptyset$);
  - literals in $\mathcal{D}$ label predecessors of c_2 (if $\mathcal{D} = \emptyset$, there is only one predecessor and it is labeled $\top$).

The subgraph containing such information can be represented schematically as shown on the left side of Figure 4, where $[\mathcal{A}]$, $[\mathcal{B}]$, $[\mathcal{C}]$ and $[\mathcal{D}]$ are the subgraphs that contains nodes whose labels are in $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$. The basic transformation step changes the subgraph in a way that the label of node v becomes $\overline{L}$. Consistently the two instances referred to above are rearranged to become the following logically equivalent formulas:

– $\mathcal{B} \rightsquigarrow (\{\overline{L}\} \cup \mathcal{A})$,
– $(\{\overline{L}\} \cup \mathcal{D}) \rightsquigarrow \mathcal{C}$.

The right side of Figure 4 shows a schema of the corresponding subgraph obtained via the basic transformation step as follows.

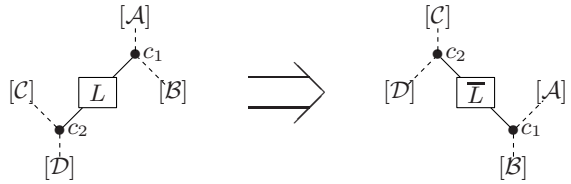


Fig. 4. Basic transformation step schema.

(Basic transformation step) Let c_1 labeled with a clause that has $(\{L\} \cup \mathcal{B}) \rightsquigarrow \mathcal{A}$ as an instance, c_2 labeled with a clause that has $\mathcal{D} \rightsquigarrow (\mathcal{C} \cup \{L\})$ as an instance and v labeled L be such as discussed above. Then:

1. eliminate v, reinsert it as successor of c_1 and predecessor of c_2 and change its label to $\overline{L}$;
2. if $\mathcal{A} = \emptyset$, eliminate the successor of c_1 labeled $\bot$;
3. if $\mathcal{B} = \emptyset$, create a predecessor for c_1 labeled $\top$;
4. if $\mathcal{C} = \emptyset$, create a successor for c_2 labeled $\bot$;
5. if $\mathcal{D} = \emptyset$, eliminate the predecessor of c_2 labeled $\top$.

Theorem 2. *An application of the basic transformation step to a proof graph results in a proof graph.*

Proof. First note that with the exception of the subgraphs involving nodes c_1 and c_2 and its successors and predecessors, all the rest remain unmodified.

Consider first the modifications related to c_1. If subgraph $[\mathcal{A}]$ is a single node labeled $\bot$, then Definition 2 says that c_1 and its successors and predecessors have form 1 in the original graph; step 2 of the basic step eliminates that node and (a) if subgraph $[\mathcal{B}]$ is empty, step 3 creates a node labeled $\top$ turning c_1 and its successors and predecessors in a subgraph of form 2, and (b) if $[\mathcal{B}]$ is not empty, c_1 and its successors and predecessors become a subgraph of form 3. On the other hand, if $[\mathcal{A}]$ is not a single node labeled $\bot$, then Definition 2 says that c_1 and its successors and predecessors have form 3 in the original graph; in the case that (a) $[\mathcal{B}]$ is empty, step 3 of the basic step creates a node labeled $\top$ turning c_1 and its successors and predecessors in a subgraph of form 2, but (b) if $[\mathcal{B}]$ is not empty, c_1 and its successors and predecessors remain a subgraph of form 3. The instances of the clauses labeling c_1 in the original and in the new graph are the same, as $(\{L\} \cup \mathcal{B}) \rightsquigarrow \mathcal{A} \equiv \mathcal{B} \rightsquigarrow (\{\overline{L}\} \cup \mathcal{A})$.

Similarly one can show that an application of the basic transformation step turns c_2 and its successors and predecessors in a subgraph of the form 1 or 3. □

Figure 5 shows the graph resulting from the application of the basic transformation step to the graph of Figure 3, where c_1 is the node labeled with clause (1), c_2 is the node labeled with clause (6) and v is the node labeled ¬ staff(J); the complement of this last literal is colored gray in Figure 5.

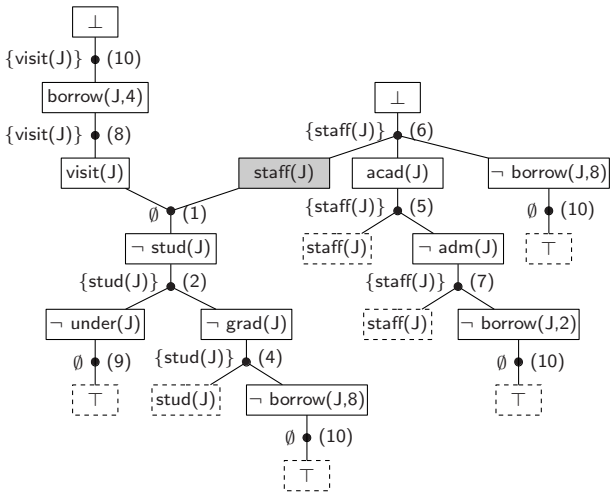


Fig. 5. A proof graph obtained from that of Figure 3.

Figure 5 also shows the support of each clause node of the proof graph. This concept had to be generalized accordingly to account for the possibility of more than one consequent node. The definition follows.

Definition 3. *The* support *of a clause node v, $S(v)$, is defined as follows.*

- *If v has one antecedent node labeled $\top$, $S(v) = \emptyset$;*
- *If v has n antecedent nodes labeled with reduced literals $L_1, \ldots, L_n$, $n \geq 1$, and no more antecedent nodes, $S(v) = \{L_1, \ldots, L_n\}$;*
- *If v has n antecedent nodes labeled with reduced literals $L_1, \ldots, L_n$, $n \geq 0$, and it has k antecedent nodes labeled with expanded literals $p_1 : M_1, \ldots, p_k : M_k$, $k \geq 1$, and $u_1, \ldots, u_k$ are the clause nodes with consequent nodes $p_1, \ldots, p_k$, then $S(v) = \{L_1, \ldots, L_n\} \cup \bigcup_{1 \leq i \leq k} h(p_i)$, where*

$$h(p_i) = \begin{cases} S(u_i) - \{\overline{M_i}\}, & \text{if } p_i \text{ is the only consequent of } u_i; \\ \{M_i\}, & \text{otherwise.} \end{cases}$$

□

A theorem similar to Theorem 1 holds for proof graphs.

Theorem 3. *Let Σ be a set of clauses from which a proof tree T is constructed, G be a proof graph constructed from T and $N_1, \ldots, N_m$ be the labels of the consequent nodes of a clause node v of G. Then $\Sigma \models disj(\overline{S(v)} \cup \{N_1, \ldots, N_m\})$.*

Proof. By structural induction. First, if v has an antecedent node labeled $\top$, then $S(v) = \emptyset$. Thus, $disj(\overline{S(v)} \cup \{N_1, \ldots, N_m\}) = N_1 \vee \ldots \vee N_m$ and this is an instance of the clause labeling v. Therefore, $\Sigma \models disj(\overline{S(v)} \cup \{N_1, \ldots, N_m\})$ holds. Second, if v has n antecedent nodes labeled with reduced literals $L_1, \ldots, L_n$, $n \geq 1$, and only those antecedent nodes, $S(v) = \{L_1, \ldots, L_n\}$ and $disj(\overline{S(v)} \cup \{N_1, \ldots, N_m\}) = \overline{L_1} \vee \ldots \vee \overline{L_n} \vee N_1 \vee \ldots \vee L_m$, a clause which is an instance of that labeling v. Again, $\Sigma \models disj(\overline{S(v)} \cup \{N_1, \ldots, N_m\})$ holds. Now consider a subgraph in which v has n antecedent nodes labeled with reduced literals $L_1, \ldots, L_n$, $n \geq 0$, and k antecedent nodes labeled with expanded literals $p_1 : M_1, \ldots, p_k : M_k$, $k \geq 1$. In this case, $S(v) = \{L_1, \ldots, L_n\} \cup \bigcup_{1 \leq i \leq k} h(p_i)$, where if $u_1, \ldots, u_k$ are the clause nodes whose consequent nodes are $p_1, \ldots, p_k$, then $h(p_i) = S(u_i) - \{\overline{M_i}\}$, if p_i is the only successor of u_i, and $h(p_i) = \{M_i\}$, if p_i is not the only successor of u_i. In the former case, the inductive hypothesis says that $\Sigma \models disj(\overline{S(u_i)} \cup \{M_i\})$. Since $\overline{S(u_i)} \cup \{M_i\} = (\overline{S(u_i)} - \{M_i\}) \cup \{M_i\} = \overline{h(p_i)} \cup \{M_i\}$ it follows that $\Sigma \models disj(\overline{h(p_i)} \cup \{M_i\})$. As $\overline{L_1} \vee \ldots \vee \overline{L_n} \vee \overline{M_1} \vee \ldots \vee \overline{M_k} \vee N_1 \vee \ldots \vee N_m$ is an instance of the clause (from Σ) that labels v, $\Sigma \models \overline{L_1} \vee \ldots \vee \overline{L_n} \vee \overline{M_1} \vee \ldots \vee \overline{M_k} \vee N_1 \vee \ldots \vee N_m$. Since for $1 \leq i \leq k$, $\Sigma \models disj(\overline{h(p_i)} \cup \{M_i\})$ if p_i is the only successor of u_i (as shown above), and $\overline{h(p_i)} = \{\overline{M_i}\}$ if p_i is not the only successor of u_i (by Definition 3), one can conclude $\Sigma \models disj(\{\overline{L_1}, \ldots, \overline{L_n}\} \cup \overline{h(p_1)} \cup \ldots \cup \overline{h(p_k)} \cup \{N_1, \ldots, N_m\}$. In other words, $\Sigma \models disj(\overline{S(v)} \cup \{N_1, \ldots, N_m\})$, as required. □

Several conclusions can be drawn from Figure 5 by applying Theorem 3. In particular, it can be seen that:

- $\Sigma \cup \{\neg\ \mathsf{borrow(J,4)}\} \models disj(\{\neg \mathsf{visit(J)}\} \cup \{\bot\})$ or equivalently $\Sigma \models \neg\ \mathsf{visit(J)} \vee \mathsf{borrow(J,4)}$.
- $\Sigma \cup \{\neg\ \mathsf{borrow(J,2)}, \neg\ \mathsf{borrow(J,8)}\} \models disj(\{\neg \mathsf{staff(J)}\} \cup \{\bot\})$ or equivalently $\Sigma \models \neg \mathsf{staff(J)} \vee \mathsf{borrow(J,2)} \vee \mathsf{borrow(J,8)}$.
- $\Sigma \cup \{\neg\ \mathsf{borrow(J,8)}\} \models disj(\emptyset \cup \{\mathsf{visit(J)}, \mathsf{staff(J)}\})$ or equivalently $\Sigma \models \mathsf{visit(J)} \vee \mathsf{staff(J)} \vee \mathsf{borrow(J,8)}$.

From these three statements the disjuntive answer follows, and it also follows the following "tentative" justificatory answer:

$\mathsf{visit(J)} \rightarrow \mathsf{borrow(J,4)}$.
$\mathsf{staff(J)} \rightarrow (\mathsf{borrow(J,2)} \vee \mathsf{borrow(J,8)})$.
$\neg\ \mathsf{visit(J)} \wedge \neg\ \mathsf{staff(J)} \rightarrow \mathsf{borrow(J,8)}$.

This answer has two problems: (*i*) it is not one of the more precise answers: there are others that eliminate the imprecision of the second statement above; (*ii*) it is not imediately recoverable from the graph: some extra reasoning was necessary to uncover it. These problems are solved by the construction of a special proof graph, an *answer proof graph*, to be introduced in the next section.

4 Arriving At Justificatory Answers

Before defining answer proof graph, a precise set of requirements for justificatory answers are presented. Let Σ be a knowledge base, $q(X)$ a question to answer

from Σ and $q(T_1) \vee \ldots \vee q(T_k)$ a disjunctive answer. A *justificatory answer* to $q(X)$ is a set of conditionals $\{\alpha_i \rightarrow \beta_i \,|\, 1 \leq i \leq m\}$ satisfying the following requirements:

- $\Sigma \models \alpha_i \rightarrow \beta_i$ for $1 \leq i \leq m$;
- each α_i is a conjunction of one or more literals, none of them in $\{q(T_1), \ldots, q(T_k)\}$;
- each β_i is a disjunction of one or more literals from $\{q(T_1), \ldots, q(T_k)\}$; and
- the set of literals of $\beta_1, \ldots, \beta_m$ is $\{q(T_1), \ldots, q(T_k)\}$.

Next, it is presented the definition of an answer proof graph.

Definition 4. *An* answer proof graph *is (a) a proof tree without occurrences of the answer predicate in its labels (reflecting the unsatisfiability of the knowledge base) or is (b) a proof graph in which every occurrence of the answer predicate occurs in a positive literal that labels a son of a root, and no root has as its son a node labeled with a literal that hasn't the answer predicate as its predicate symbol.*

Starting from the proof tree of Figure 3, it is possible to arrive at the answer proof graph of Figure 6 after a certain number of applications of the basic transformation step. Note that there is one root for each occurrence of the predicate symbol borrow in the initial proof tree.

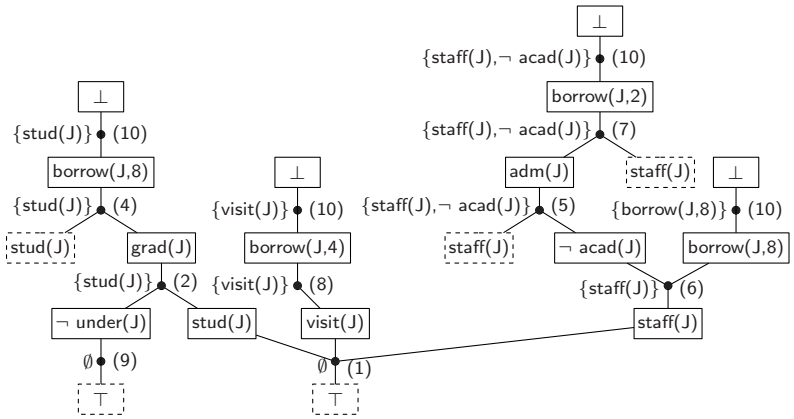


Fig. 6. Example of answer proof graph with supports.

From the supports of the clause nodes whose consequent nodes have labels with the predicate symbol borrow in the proof graph of Figure 6, taken from left to right, one can state the following consequences of the knowledge base of Example 2:

1. stud(J) $\rightarrow$ borrow(J,8)
2. visit(J) $\rightarrow$ borrow(J,4)
3. staff(J) $\wedge\neg$ acad(J) $\rightarrow$ borrow(J,2)
4. staff(J) $\rightarrow \neg$ acad(J) $\vee$ borrow(J,8)

The first three consequences are exactly in the format stated in the requirements. The last one can be put in that format by using one of its contrapositives:

4'. $\mathsf{staff(J)} \wedge \mathsf{acad(J)} \rightarrow \mathsf{borrow(J,8)}$

This is always possible, obviously, when the consequent has only one literal with the occurrence of the answer predicate. Let G be an answer proof graph of the kind described in Definition 4(b). Let v be a clause node that has at least one consequent node whose label is a literal on the answer predicate. Then, for each such clause node v, the following "answer" follows from the knowledge base:

$$S(v) \cup \{\overline{M_1}, \ldots, \overline{M_k}\} \rightsquigarrow \{q(T_1), \ldots, q(T_l)\}$$

where M_1, ..., M_k, $q(T_1)$, ..., $q(T_l)$ are labels of all consequent nodes of v, the first k don't contain the answer predicate q, $k \geq 0$, and the rest contain occurrences of q, $l \geq 1$. According to Theorem 3, $\Sigma \models \mathit{disj}(\overline{S(v)} \cup \{M_1, \ldots, M_k\} \cup \{q(T_1), \ldots, q(T_l)\})$, and this last formula is equivalent to $S(v) \cup \{\overline{M_1}, \ldots, \overline{M_k}\} \rightsquigarrow \{q(T_1), \ldots, q(T_l)\}$. These answers have exactly the required form for a justificatory answer.

5 Conclusion

There are situations in which a question asks for information about an individual, but the answer to this question depends on the specific characteristics of that individual that are incomplete or even absent from the knowledge base. Usually in such situations, only disjunctive answers can be obtained.

In this paper, we addressed this problem in the context of a first-order knowledge base. We considered questions of the form *find x such that $q(X)$*, where q is a predicate symbol and X is an n-tuple of variables, that result in a disjunctive answer $q(T_1) \vee \ldots \vee q(T_k)$, $k \geq 2$, where each T_i is an n-tuple of terms. A disjunctive answer of this form, although correct, may not be informative enough. The reason is that, when supplied with such answer, the user may not be able to determine which disjunct $q(T_i)$, $1 \leq i \leq k$, is the appropriate answer to his question.

As a solution, we presented a method that allows, from a proof tree corresponding to a deduction of a disjunctive answer $q(T_1) \vee \ldots \vee q(T_k)$, $k \geq 2$, the construction of a *justificatory answer*, an answer that gives conditions under which each disjunct $q(T_i)$ is true. Our method first transforms the specified proof tree into an answer proof graph and, then, from the resulting answer proof graph, it extracts a justificatory answer.

Several answer proof graphs can be produced from the same proof tree. Each answer proof graph may generate a different justificatory answer to the question. Thus, we must make here a last remark regarding the *quality* of answers, as it could be possible that it is not the same for all answer proof graphs. It could be the case that some of them supply answers more meaningful than others. In reality, the meaningfulness of an answer could depend on characteristics of the user not present in a proof tree (and consequently not in an proof graph).

Typically, one would try to capture such characteristics in a user model. Any way, the method presented in this paper will not exclude any of the possibly meaningful justificatory answers.

As future work, we intend to generalize our method to the other classes of answers (generic and hypothetical answers).

References

1. Barbosa, Isabel G., Vieira, N.J.: Extracting case-based answers from closed proof-trees. In: Proceedings of the 2nd International Conference on Agents and Articial Intelligence, Volume 1, 2010.
2. Brachman, R., Levesque, H.: Knowledge Representation and Reasoning (The Morgan Kaufmann Series in Artificial Intelligence). Morgan Kaufmann, Amsterdam; Boston, 2004.
3. Bruynooghe, M.: The Memory Management of Prolog Implementations. In K. L. Clark, S.-A. Tarnlund, eds.: Logic Programming, pp. 83-98, Academic Press, London, 1982.
4. Burhans, D.T. , Shapiro, S.C.: Defining Answer Classes Using Resolution Refutation. J. Applied Logic, 5(1):70-91, 2007.
5. Chang, C.-L., Lee, R.C.-T.: Symbolic Logic and Mechanical Theorem Proving. Academic Press, Inc., Orlando, FL, USA, 1973.
6. Demolombe, R.: A Strategy for the Computation of Conditional Answers. In: ECAI 92: Proceedings of the 10th European Conference on Artificial intelligence, pp. 134-138, New York, NY, USA, 1992.
7. Green, C.: The Application of Theorem Proving to Question-Answering Systems. PhD thesis, Department of Electrical Engineering, Stanford University, 1969.
8. Letz, R., Stenz, G.: Model Elimination and Connection Tableau Procedures. In: Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Volume II. pp. 2015-2112, Elsevier Science, 2001.
9. Loveland, D.W.: A Simplified Format for the Model Elimination Theorem-Proving Procedure. Journal of the ACM, 16(3):349-363, 1969.
10. Loveland, D.W.: Automated Theorem Proving: A Logical Basis. North-Holland, 1978.
11. Loveland, D.W., Stickel, M.E.: A Hole in Goal Trees: Some Guidance from Resolution Theory. In: IJCAI 73: Proceedings of the 3rd International Joint Conference on Artificial intelligence, pp. 153-161, Stanford, CA, USA, 1973.
12. Luckham, D., Nilsson, N.J.: Extracting Information from Resolution Proof Trees. Artificial Intelligence, 2:27-54, 1971.
13. van Emden, M.H.: An Interpreting Algorithm for Prolog Programs. In J.A. Campbell, ed.: Implementations of Prolog, pp. 93-110, Ellis Horwood, 1984.
14. Vieira, N.J.: Máquinas de Inferência para Sistemas Baseados em Conhecimento. PhD thesis, PUC/RJ, 1987 (in Portuguese).