Algorithms for effective argumentation in classical propositional logic: A connection graph approach

Vasiliki Efstathiou, Anthony Hunter

Department of Computer Science
University College London
Gower Street, London WC1E 6BT, UK
{v.efstathiou, a.hunter}@cs.ucl.ac.uk

Abstract There are a number of frameworks for modelling argumentation in logic. They incorporate a formal representation of individual arguments and techniques for comparing conflicting arguments. A common assumption for logic-based argumentation is that an argument is a pair $\langle \Phi, \alpha \rangle$ where Φ is minimal subset of the knowledgebase such that Φ is consistent and Φ entails the claim α. Different logics provide different definitions for consistency and entailment and hence give us different options for argumentation. Classical propositional logic is an appealing option for argumentation but the computational viability of generating an argument is an issue. Here we propose ameliorating this problem by using connection graphs to give information on the ways that formulae of the knowledgebase can be used to minimally and consistently entail a claim. Using a connection graph allows for a substantially reduced search space to be used when seeking all the arguments for a claim from a knowledgebase. We provide a theoretical framework and algorithms for this proposal, together with some theoretical results and some preliminary experimental results to indicate the potential of the approach.

1 Introduction

Argumentation is a vital aspect of intelligent behaviour by humans. Consider diverse professionals such as politicians, journalists, clinicians, scientists, and administrators, who all need to collate and analyse information looking for pros and cons for consequences of importance when attempting to understand problems and make decisions.

There are a number of proposals for logic-based formalisations of argumentation (for reviews see [9,21,5]). These proposals allow for the representation of arguments for and against some claim, and for counterargument relationships between arguments. In a number of key examples of argumentation systems, an argument is a pair where the first item in the pair is a minimal consistent set of formulae that proves the second item which is a formula (see for example [2,14,3,1,15,4]). Furthermore, in these approaches, a key form of counterargument is an undercut: One argument undercuts another argument when the claim of the first argument negates the premises of the second argument. Proof procedures and algorithms have been developed for finding preferred arguments from a knowledgebase using defeasible logic and following for example Dung's preferred semantics (see for example [7,23,20,17,8,11,12]). However, these techniques and

analyses do not offer any ways of ameliorating the computational complexity inherent in finding arguments and counterarguments for classical logic. Furthermore, we wish to find all arguments for a particular claim, and this means a pruning strategy, such as incorporated into defeasible logic programming [15,10], would not meet our requirements since some undercuts would not be obtained.

In this paper we restrict the language used to a language of (disjunctive) clauses and for this language we propose algorithms for finding arguments using search tree structures that correspond to the steps of a systematic application of the connection graph proof procedure [18,19]. We describe how this method can be efficient regarding the computational cost of finding arguments.

2 Logical argumentation for a language of clauses

In this section, we adapt an existing proposal for logic-based argumentation [3] by restricting the language to being disjunctive clauses so that the premises of an argument is a set of clauses and the claim of an argument is a literal.

Definition 1. *A language of clauses $\mathcal{C}$ is composed from a set of atoms $\mathcal{A}$ as follows: If α is an atom, then α is a* **positive literal***, and $\neg\alpha$ is a* **negative literal***. If β is a positive literal, or β is a negative literal, then β is a* **literal***. If $\beta_1, .., \beta_n$ are literals, then $\beta_1 \vee ... \vee \beta_n$ is a* **clause***. A* **clause knowledgebase** *is a set of clauses.*

We use $\phi, \psi, \ldots$ to denote disjunctive clauses and $\Delta, \Phi, \Psi, \ldots$ to denote sets of clauses. For the following definitions, we first assume a clause knowledgebase Δ (a finite set of clauses) and use this Δ throughout. The paradigm for the approach is that there is a large repository of information, represented by Δ, from which arguments can be constructed for and against arbitrary claims. Apart from information being understood as declarative statements, there is no *a priori* restriction on the contents, and the pieces of information in the repository can be arbitrarily complex. Therefore, Δ is not expected to be consistent.

The framework adopts a very common intuitive notion of an argument. Essentially, an argument is a set of clauses that can be used to prove some claim, together with that claim. In this paper, we assume each claim is represented by a literal.

Definition 2. *A* **literal argument** *is a pair $\langle \Phi, \alpha \rangle$ such that: (1) α is a literal (2) $\Phi \subseteq \Delta$; (3) $\Phi \nvdash \bot$; (4) $\Phi \vdash \alpha$; and (5) there is no $\Phi' \subset \Phi$ such that $\Phi' \vdash \alpha$. We say that $\langle \Phi, \alpha \rangle$ is a literal argument for α. We call α the* **claim** *of the argument and Φ the* **support** *of the argument (we also say that Φ is a support for α).*

Example 1. Let $\Delta = \{a, \neg a \vee b, \neg b \vee c, b \vee \neg d, \neg a, a \vee b, \neg c, \neg b \vee \neg c, c \vee a\}$. Some literal arguments are:

$$\langle \{a, \neg a \vee b\}, b \rangle$$
$$\langle \{\neg a\}, \neg a \rangle$$
$$\langle \{a, \neg a \vee b, \neg b \vee c\}, c \rangle$$
$$\langle \{a \vee b, \neg b \vee \neg c, c \vee a\}, a \rangle$$

Some arguments oppose the claim or the support of other arguments. This leads to the notion of a counterargument as follows.

Definition 3. *Let $\langle \Phi, \alpha \rangle$ and $\langle \Psi, \beta \rangle$ be literal arguments*

- *$\langle \Psi, \beta \rangle$ is a* **rebut** *of $\langle \Phi, \alpha \rangle$ iff $\{\beta, \alpha\} \vdash \bot$.*
- *$\langle \Psi, \beta \rangle$ is an* **undercut** *of $\langle \Phi, \alpha \rangle$ iff $\Phi \vdash \neg\beta$.*
- *$\langle \Psi, \beta \rangle$ is a* **counteragument** *of $\langle \Phi, \alpha \rangle$ iff $\langle \Psi, \beta \rangle$ is a rebut or an undecut of $\langle \Phi, \alpha \rangle$.*

Example 2. $\langle \{\neg c \vee b, c\}, b \rangle$ is a rebut of $\langle \{\neg a, a \vee \neg d, d \vee \neg b \vee c, \neg c\}, \neg b \rangle$.
$\langle \{\neg c \vee b, c\}, b \rangle$ is an undercut of $\langle \{a, d, \neg a \vee \neg d \vee \neg b, b \vee e\}, e \rangle$.

Following a number of proposals for argumentation (e.g. [3,1,15,20,12]), logical arguments and counterarguments can be presented in a graph: Each node denotes an argument and each arc (A_1, A_2) denotes that argument A_2 is a counterargument to argument A_1. Various constraints have been imposed on the nature of such graphs leading to a range of options for evaluating whether a particular argument in the graph is "defeated" or "undefeated". We will not consider this aspect of logic-based argumentation further in this paper. We are only concerned in this paper with how we can construct the arguments from the knowledgebase, and not how to compare them.

3 Towards effective algorithms for generating arguments

We now turn to automating the construction of arguments and counterarguments. Unfortunately automated theorem proving technology cannot do this directly for us. For each argument, we need a minimal and consistent set of formulae that proves the claim. An automated theorem prover (an ATP) may use a "goal-directed" approach, bringing in extra premises when required, but they are not guaranteed to be minimal and consistent. For example, supposing we have a clause knowledgebase $\{\neg\alpha \vee \beta, \beta\}$, for proving β, the ATP may start with the premise $\neg\alpha \vee \beta$, then to prove β, a second premise is required, which would be β, and so the net result is $\{\neg\alpha \vee \beta, \beta\}$, which does not involve a minimal set of premises. In addition, an ATP is not guaranteed to use a consistent set of premises since by classical logic it is valid to prove anything from an inconsistency.

So if we seek arguments for a particular claim δ, we need to post queries to an ATP to ensure that a particular set of premises entails δ, that the set of premises is minimal for this, and that it is consistent. So finding arguments for a claim α involves considering subsets Φ of Δ and testing them with the ATP to ascertain whether $\Phi \vdash \alpha$ and $\Phi \nvdash \bot$ hold. For $\Phi \subseteq \Delta$, and a formula α, let $\Phi?\alpha$ denote a call (a query) to an ATP. If Φ classically entails α, then we get the answer $\Phi \vdash \alpha$, otherwise we get the answer $\Phi \nvdash \alpha$. In this way, we do not give the whole of Δ to the ATP. Rather we call it with particular subsets of Δ. So for example, if we want to know if $\langle \Phi, \alpha \rangle$ is an argument, then we have a series of calls $\Phi?\alpha$, $\Phi?\bot$, $\Phi \setminus \{\phi_1\}?\alpha$,...,$\Phi \setminus \{\phi_k\}?\alpha$, where $\Phi = \{\phi_1, .., \phi_k\}$. So the first call is to ensure that $\Phi \vdash \alpha$, the second call is to ensure that $\Phi \nvdash \bot$, the remaining calls are to ensure that there is no subset Φ' of Φ such that $\Phi' \vdash \alpha$. This then raises the question of which subsets Φ of Δ to investigate to determine whether $\langle \Phi, \alpha \rangle$ holds when we are seeking for an argument for α.

A further problem we need to consider is that if we want to generate all arguments for a particular claim in the worst case we may have to send each subset Φ of Δ to the ATP to determine whether $\Phi \vdash \alpha$ and $\Phi \nvdash \bot$. So in the worst case, if $|\Delta| = n$, then we may need to make 2^{n+1} calls to the ATP. Even for a small knowledgebase of say 20 or 30 formulae, this can become prohibitively expensive.

It is with these issues in mind that we explore an alternative way of finding all the arguments from a knowledgebase Δ for a claim α. Our approach is to adapt the idea of connection graphs to enable us to find arguments.

4 Connection graphs

Connection graphs were initially proposed by Kowalski (see [18,19]) for reducing the search space for applying resolution to clauses in logic programming. They have also been developed more generally for classical logic [6]. In this section we will adapt the definition of a connection graph to give us the notion of a focal graph which for a knowledgebase Δ and a claim α essentially delineates the subset of the knowledgebase that may have a role in an argument for the claim α. For example, for $\Delta = \{a, \neg a \vee f, \neg a \vee b, \neg b \vee c, \neg n \vee \neg m, b \vee d, b \vee e, \neg e \vee a, \neg d \vee a \vee \neg c, \neg g \vee m, \neg q \vee r \vee p, \neg p\}$ and the claim a we require that the delineated subset is $\{a, \neg a \vee b, \neg b \vee c, b \vee d, b \vee e, \neg e \vee a, \neg d \vee a \vee \neg c\}$. In this way, formulae that cannot possibly be a premise in an argument will be excluded. This provides the potential for substantially reducing the set of formulae to be considered for constructing arguments.

So in this section we will formalize the notion of a focal graph, then in the next section we consider how we can search the focal graph, and in the subsequent section we provide algorithms for efficiently searching the focal graph so as to return all the arguments for the claim of interest.

We start by introducing some relations on the elements of $\mathcal{C}$, that will be used to determine the links of the connection graphs and how these can be used by the search algorithms.

Definition 4. *The* Disjuncts *function takes a clause and returns the set of disjuncts in the clause*

$$\mathsf{Disjuncts}(\beta_1 \vee .. \vee \beta_n) = \{\beta_1, .., \beta_n\}$$

Definition 5. *Let ϕ and ψ be clauses. Then,* $\mathsf{Preattacks}(\phi, \psi) = \{\beta \mid \beta \in \mathsf{Disjuncts}(\phi)$ *and* $\neg\beta \in \mathsf{Disjuncts}(\psi)\}$.

Example 3. $\mathsf{Preattacks}(a \vee \neg b \vee \neg c \vee d, a \vee b \vee \neg d \vee e) = \{\neg b, d\}$, $\mathsf{Preattacks}(a \vee b \vee \neg d \vee e, a \vee \neg b \vee \neg c \vee d) = \{b, \neg d\}$, $\mathsf{Preattacks}(a \vee b \vee \neg d, a \vee b \vee c) = \emptyset$.

Definition 6. *Let ϕ and ψ be clauses. If* $\mathsf{Preattacks}(\phi, \psi) = \{\beta\}$ *for some β, then* $\mathsf{Attacks}(\phi, \psi) = \beta$ *otherwise* $\mathsf{Attacks}(\phi, \psi) = null$.

Example 4. $\mathsf{Attacks}(a \vee \neg b \vee \neg c \vee d, a \vee b \vee \neg d \vee e) = null$, $\mathsf{Attacks}(a \vee b \vee \neg d, a \vee b \vee c) = null$, $\mathsf{Attacks}(a \vee b \vee \neg d, a \vee b \vee d) = \neg d$, $\mathsf{Attacks}(a \vee b \vee \neg d, e \vee c \vee d) = \neg d$.

Hence, the Preattacks relation is defined for any pair of clauses ϕ, ψ while the Attacks relation is defined for a pair of clauses ϕ, ψ such that $|\mathsf{Preattacks}(\phi, \psi)| = 1$

Lemma 1. *We can see that for two clauses ϕ and ψ if* $\mathsf{Preattacks}(\phi, \psi) \neq \emptyset$ *then from the resolution proof rule it follows that* $\forall \beta \in \mathsf{Preattacks}(\phi, \psi)$*,*

$$\{\phi, \psi\} \vdash \bigvee((\mathsf{Disjuncts}(\phi) \setminus \{\beta\}) \cup (\mathsf{Disjuncts}(\psi) \setminus \{\neg\beta\}))$$

Example 5. For $\phi = \neg a \vee b \vee c \vee d$ and $\psi = a \vee b \vee e \vee \neg c$, $\mathsf{Preattacks}(\phi, \psi) = \{\neg a, c\}$ hence

$$\begin{array}{l} \{\phi, \psi\} \vdash \bigvee((\mathsf{Disjuncts}(\phi) \setminus \{\neg a\}) \cup (\mathsf{Disjuncts}(\psi) \setminus \{a\})) \\ = \bigvee((\{\neg a, b, c, d\} \setminus \{\neg a\}) \cup (\{a, b, e, \neg c\} \setminus \{a\})) \\ = \bigvee(\{b, c, d, e, \neg c\}) = b \vee c \vee d \vee e \vee \neg c. \end{array}$$

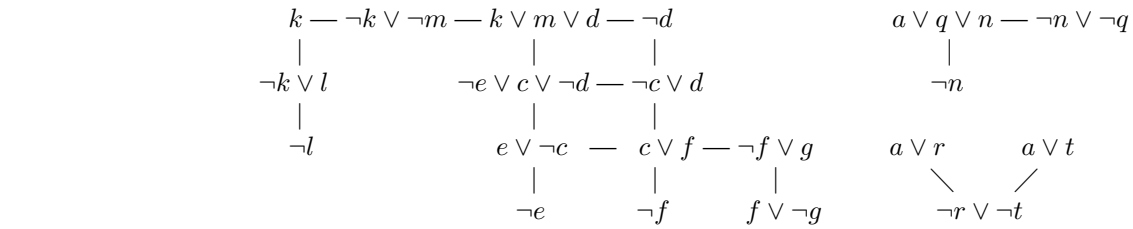
$$\begin{array}{l} \{\phi, \psi\} \vdash \bigvee((\mathsf{Disjuncts}(\phi) \setminus \{c\}) \cup (\mathsf{Disjuncts}(\psi) \setminus \{\neg c\})) \\ = \bigvee((\{\neg a, b, c, d\} \setminus \{c\}) \cup (\{a, b, e, \neg c\} \setminus \{\neg c\})) \\ = \bigvee(\{\neg a, b, d, e, a\}) = \neg a \vee b \vee d \vee e \vee a. \end{array}$$

From Lemma 1 we can see that $\mathsf{Preattacks}(\phi, \psi) \neq \mathsf{Attacks}(\phi, \psi)$ iff ϕ with ψ resolve to a tautology.

We now introduce some types of graphs whose nodes correspond to a set of clauses and the links between each pair of clauses are determined according to the attack relations defined above. In the following examples of graphs we use the $|$, $/$, $\setminus$ and $—$ symbols to denote arcs in the pictorial representation of a graph.

Definition 7. *Let Δ be a clause knowledgebase. The* **connection graph** *for Δ, denoted* $\mathsf{Connect}(\Delta)$*, is a graph (N, A) where $N = \Delta$ and $A = \{(\phi, \psi) \mid$ there is a $\beta \in \mathsf{Disjuncts}(\phi)$ such that $\beta \in \mathsf{Preattacks}(\phi, \psi)\}$.*

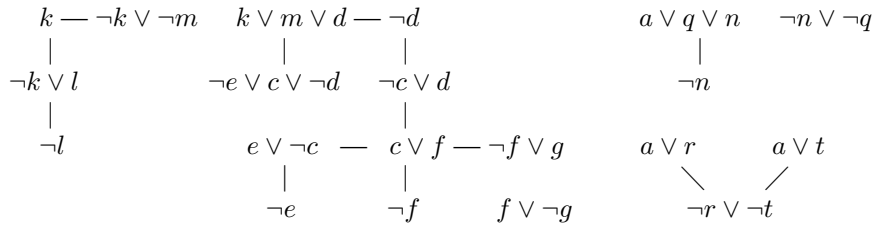
Example 6. The following is the connection graph for $\Delta = \{k, \neg k \vee l, \neg l, \neg k \vee \neg m, k \vee m \vee d, \neg d, \neg e \vee c \vee \neg d, \neg c \vee d, e \vee \neg c, c \vee f, \neg f \vee g, \neg e, \neg f, f \vee \neg g, a \vee q \vee n, \neg n \vee \neg q, \neg n, a \vee r, a \vee t, \neg r \vee \neg t\}$.



We now need to go beyond Kowalski's idea of a connection graph and introduce the following types of graph. The attack graph defined below is a subgraph of the connection graph identified using the Attacks function.

Definition 8. *Let Δ be a clause knowledgebase. The* **attack graph** *for Δ, denoted* $\mathsf{AttackGraph}(\Delta)$*, is a graph (N, A) where $N = \Delta$ and $A = \{(\phi, \psi) \mid$ there is a $\beta \in \mathsf{Disjuncts}(\phi)$ such that $\mathsf{Attacks}(\phi, \psi) = \beta\}$.*

Example 7. Continuing Example 6, the following is the attack graph for Δ.

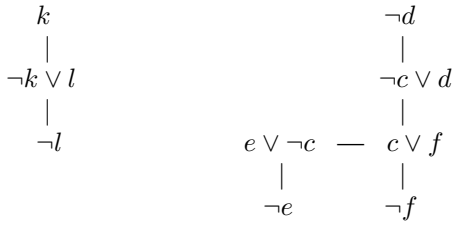


The following definition of closed graph gives a kind of connected subgraph of the attack graph where connectivity is determined in terms of the attack relation among its nodes.

Definition 9. *Let Δ be a clause knowledgebase. The* **closed graph** *for Δ, denoted* $\mathsf{Closed}(\Delta)$*, is the largest subgraph* (N, A) *of* $\mathsf{AttackGraph}(\Delta)$*, such that for each* $\phi \in N$*, for each* $\beta \in \mathsf{Disjuncts}(\phi)$ *there is a* $\psi \in N$ *with* $\mathsf{Attacks}(\phi, \psi) = \beta$*.*

The above definition assumes that there is a unique largest subgraph of the attack graph that meets the conditions presented. This is justified because having a node from the attack graph in the closed graph does not exclude any other node from the attack graph also being in the closed graph. Any subset of nodes is included when each of the disjuncts is negated by disjuncts in the other nodes. Moreover, we can consider the closed graph being composed of components where for each component Y, and for each node ϕ in Y, and for each disjunct β in ϕ, there is another node ψ in Y such that there is a disjunct $\neg\beta$ in ψ. So the nodes in each component work together to ensure each disjunct is negated by a disjunct in another node in the component, and the largest subgraph of the attack graph is obtained by just taking the union of these components.

Example 8. Continuing Example 7, the following is the closed graph for Δ.



The focal graph (defined next) is a subgraph of the closed graph for Δ which is delineated by a clause ϕ from Δ and corresponds to the part of the closed graph that contains ϕ. In the following, we assume a component of a graph means that each node in the component is connected to any other node in the component by a path.

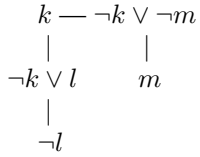
Definition 10. *Let Δ be a clause knowledgebase and ϕ be a clause in Δ. The* **focal graph** *of ϕ in Δ denoted* $\mathsf{Focal}(\Delta, \phi)$ *is defined as follows: If there is a component X in* $\mathsf{Closed}(\Delta)$ *containing the node ϕ, then* $\mathsf{Focal}(\Delta, \phi) = X$*, otherwise* $\mathsf{Focal}(\Delta, \phi)$ *is the empty graph.*

Example 9. Continuing Example 8, if $C_2 = (N_1, A_1)$ is the component of the closed graph for Δ with $N_1 = \{k, \neg k \vee l, \neg l\}$ and $C_2 = (N_2, A_2)$ is the component of the closed graph for Δ with $N_2 = \{\neg d, \neg c \vee d, e \vee \neg c, c \vee f, \neg e, \neg f\}$ then the focal graph of ϕ in Δ is C_1 for $\phi \in \{k, \neg k \vee l, \neg l\}$, and it is C_2 for $\phi \in \{\neg d, \neg c \vee d, e \vee \neg c, c \vee f, \neg e, \neg f\}$. For any other ϕ, the focal graph of ϕ in Δ corresponds to the empty graph.

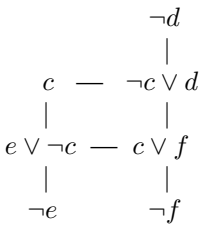
The query graph of a literal α in a clause knowledgebase Δ defined below is the graph whose elements, as we will see, determine all the literal arguments for α, if there are any.

Definition 11. *Let Δ be a clause knowledgebase and α be a literal. The* **query graph** *of α in Δ, denoted* $\mathsf{Query}(\Delta, \alpha)$*, is the focal graph of $\neg\alpha$ in $\Delta \cup \{\neg\alpha\}$. Hence,* $\mathsf{Query}(\Delta, \alpha) = \mathsf{Focal}(\Delta \cup \{\neg\alpha\}, \neg\alpha)$.

Example 10. For knowledgebase Δ given in Example 6, the following is the query graph of $\neg m$ in Δ,



and the following is the query graph of $\neg c$ in Δ.



The query graph of a literal α in a clause knowledgebase Δ delineates the subset of the Δ that contains formulae that may be a premise in a literal argument for α. Furthermore, the query graph contains information about how the formulae relate to each other in the sense of how they can potentially form proofs for the claim. Now, in order to determine whether there are any arguments that can be obtained from these formulae and to determine the support for these arguments, we need to search the query graph. This is the subject of the next section.

5 Searching query graphs

The set of nodes of the query graph of α in Δ contains all the subsets of the knowledgebase that can be used as supports for literal arguments for α. The appropriate subsets can be obtained by selecting the nodes of the query graph of α in Δ that obey certain conditions. For this we will use the notion of a support tree which represents the support set for a literal argument for α together with the negation of α in a tree structure where

$\neg\alpha$ is the root. Essentially a support tree is constructed from a subgraph of the query graph of α in Δ.

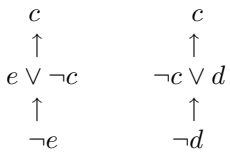
In order to define the notion of a support tree we will first introduce the notion of the presupport tree which is a tree with $\neg\alpha$ as the root and some clauses from $\mathsf{Query}(\Delta, \alpha)$ as nodes on its branches. Then we will introduce some additional constraints that define a support tree as a special kind of a presupport tree.

Definition 12. *Let Δ be a clause knowledgebase and let α be a literal. A* **presupport tree** *for Δ and α is tuple (N, A, f) where (N, A) is a tree, and f is a mapping from N to Δ such that*

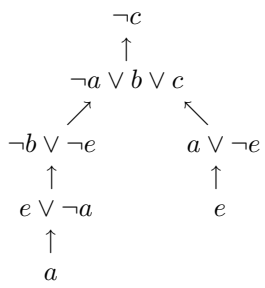
(1) *if x is the root of the tree, then $f(x) = \neg\alpha$ and there is exactly one child y of x s.t.* $\mathsf{Attacks}(f(y), f(x)) = \alpha$,
(2) *for any nodes x, y in the same branch, if $x \neq y$, then $f(x) \neq f(y)$,*
(3) *for any node x in the tree, if y is a child of x,*
then there is a $\neg\beta_i \in \mathsf{Disjuncts}(f(x))$ s.t. $\mathsf{Attacks}(f(y), f(x)) = \beta_i$
and for each $\beta_j \in \mathsf{Disjuncts}(f(y)) \setminus \{\beta_i\}$,
i) either there is exactly one child z of y s.t. $\mathsf{Attacks}(f(z), f(y)) = \neg\beta_j$,
ii) or there is an arc (w, w') in the branch containing y s.t.
$\mathsf{Attacks}(f(w), f(w')) = \beta_j$ *and w' is the parent of w.*

The first condition of the definition initialises the tree structure of the presupport tree by setting the negated claim as the clause identifying the root and ensures that it will be attacked by some other clause from the presupport tree otherwise the tree will be empty. The fact that the root can only have one child guarantees that the width of the first level of the tree will be minimized. The second condition of the definition ensures that for a finite Δ there can only be presupport trees of finite depth. A clause from Δ can have its value assigned to exactly one node in a branch ensuring that no repetitions of the same clause will be allowed in this branch. The third condition of the definition is the equivalent of condition 1 for the general case of non-root nodes. It ensures that all the disjuncts of the clause identifying a node are attacked by a node of the same branch. Each node has as many children as the number of its disjuncts that do not appear as attack values on the branch earlier, ensuring that only the necessary number of children will be in the tree at each level.

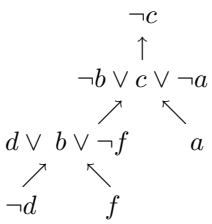
Example 11. Going back to Example 10, for $\Delta = \{k, \neg k \vee l, \neg l, \neg k \vee \neg m, k \vee m \vee d, \neg d, \neg e \vee c \vee \neg d, \neg c \vee d, e \vee \neg c, c \vee f, \neg f \vee g, \neg e, \neg f, f \vee \neg g, a \vee q \vee n, \neg n \vee \neg q, \neg n, a \vee r, a \vee t, \neg r \vee \neg t\}$ and $\alpha = \neg c$ there are two presupport trees for Δ and α.



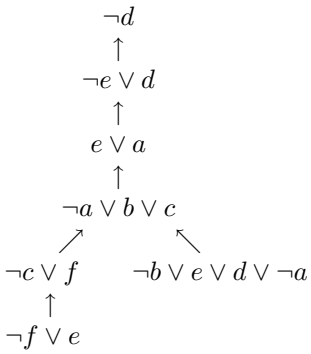
Example 12. The following is a presupport tree for $\Delta = \{\neg d, \neg a \vee b \vee c, \neg b \vee \neg e, a \vee \neg e, \neg e, e, e \vee d, e \vee \neg a, a\}$ and $\alpha = c$.



Example 13. The following is a presupport tree for $\Delta = \{a, \neg c, \neg b \vee c \vee \neg a, \neg d, b, \neg e, d \vee b \vee \neg f, \neg b, f\}$ and $\alpha = c$.



Example 14. The following is a presupport tree for $\Delta = \{\neg e \vee d, e \vee a, \neg a \vee b \vee c, \neg c \vee f, \neg b \vee e \vee d \vee \neg a, \neg f \vee e, \neg a \vee g, \neg g \vee h\}$ and $\alpha = d$.



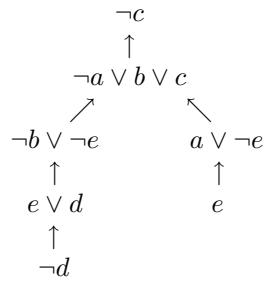
Proposition 1. *If (N, A, f) is a presupport tree for a finite clause knowledgebase Δ and α then (N, A) is a finite tree.*

We will now introduce two special cases of presupport trees each of which amounts to the notions of minimal entailment and consistent entailment.

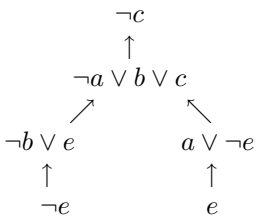
Definition 13. *Let Δ be a clause knowledgebase and let α be a literal. A* **consistent presupport tree** *for Δ and α is a presupport tree (N, A, f) for Δ and α such that for any nodes x and y where x' is the parent of x and y' is the parent of y,* $\mathsf{Attacks}(f(x), f(x')) \neq \neg\mathsf{Attacks}(f(y), f(y'))$.

So, a presupport tree is consistent if does not contain any pair of arcs $(x, x'), (y, y')$ such that $\mathsf{Attacks}(f(x), f(x')) = \neg\beta$ and $\mathsf{Attacks}(f(y), f(y')) = \beta$ for some $\neg\beta \in \mathsf{Disjuncts}(f(x))$ and $\beta \in \mathsf{Disjuncts}(f(y))$.

Example 15. The following is a consistent presupport tree for $\Delta = \{\neg d, \neg a \vee b \vee c, \neg b \vee \neg e, a \vee \neg e, \neg e, e, e \vee d, \neg d\}$ and $\alpha = c$.



Example 16. The following is not a consistent presupport tree for $\Delta = \{\neg d, \neg a \vee b \vee c, \neg b \vee e, a \vee \neg e, \neg e, e\}$ and $\alpha = c$: for $f(x') = \neg b \vee e, f(x) = \neg e, f(y') = a \vee \neg e, f(y) = e$ we get $\mathsf{Attacks}(f(x), f(x')) = \neg e$ and $\mathsf{Attacks}(f(y), f(y')) = e$.



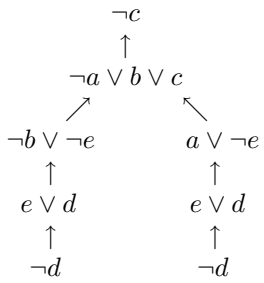
Definition 14. *Let Δ be a clause knowledgebase and let α be a literal. A* **minimal presupport tree** *for Δ and α is a presupport tree (N, A, f) for Δ and α such that:*

(1) *for any nodes x, y in the same branch where x' is the parent of x and y' is the parent of y*
$\mathsf{Attacks}(f(x), f(x')) \neq \mathsf{Attacks}(f(y), f(y'))$
(2) *if for two nodes x and y, where x' is the parent of x and y' is the parent of y,*
$\mathsf{Attacks}(f(x), f(x')) = \mathsf{Attacks}(f(y), f(y'))$
then $\mathsf{Subtree}(x) \subseteq \mathsf{Subtree}(y')$ *or* $\mathsf{Subtree}(y) \subseteq \mathsf{Subtree}(x')$

Where $\mathsf{Subtree}(x)$ *is the set of formulae in the subtree rooted at x.*

The first condition of this definition ensures that nodes that are not necessary for the entailment of the given claim cannot be added on the branches of a minimal presupport tree. The second condition ensures that if two nodes x and y need to be attacked on the same disjunct then common nodes will be used to attack both, ensuring that there will be no more than one set of nodes contributing to the entailment of the claim in the same way.

Example 17. The presupport tree of Example 16 is a minimal presupport tree for Δ and α. The presupport tree of Example 15 is not a minimal presupport tree for Δ and α because it violates the second condition of Definition 14. If in the presupport tree of Example 15 we replace $\mathsf{Subtree}(x_2)$ by a copy of $\mathsf{Subtree}(x_1)$ for x_1, x_2 such that $f(x_1) = e \vee d$ and $f(x_2) = e$ then both conditions of the definition will be satisfied and we will obtain the following minimal presupport tree for Δ and α:



Definition 15. *A presupport tree (N, A, f) is a* **support tree** *iff it is a minimal and consistent presupport tree.*

Example 18. Each of the presupport trees of Examples 11, 13 and 14 is a support tree.

We will now introduce some theoretical results illustrating why support trees can be useful for our purposes in seeking arguments for a claim from a knowledgebase. First we give the definition of a minimal inconsistent subset of a knowledgebase Δ and then we give a proposition illustrating how these sets can be used in argumentation and how they relate to support trees.

Definition 16. *For a set of formulae Δ, a* **minimal inconsistent subset** *Φ of Δ is such that:*

(1) $\Phi \vdash \bot$
(2) *For all* $\Psi \subseteq \Delta$, *if* $\Psi \subset \Phi$, *then* $\Psi \nvdash \bot$.

Proposition 2. *For a literal α, $\langle \Phi, \alpha \rangle$ is a literal argument iff $\Phi \cup \{\neg\alpha\}$ is a minimal inconsistent subset of $\Delta \cup \{\neg\alpha\}$.*

Proposition 3. *If (N, A, f) is a support tree for Δ and α, and $\Gamma = \{f(x) \mid x \in N\}$, then $\Gamma \vdash \bot$ and for any $\Gamma' \subset \Gamma, \Gamma' \nvdash \bot$.*

From proposition 3 we get that the clause knowledgebase that corresponds to a support tree for Δ and α is a minimal inconsistent set and hence the following proposition holds.

Proposition 4. *If (N, A, f) is a support tree for Δ and α, then $\{f(x) \mid x \in N\} \setminus \{\neg\alpha\} \vdash \alpha$.*

According to the following proposition, the clause knowledgebase that corresponds to the nodes of a support tree for Δ and α cannot contain another knowledgebase that can be arranged is a support tree structure for Δ and α.

Proposition 5. *If (N, A, f) is a support tree for Δ and α and (N', A', f') is a support tree for Δ and α , then $\{f(x) \mid x \in N\} \not\subset \{f'(x') \mid x' \in N'\}$.*

From the last four propositions it follows that for any minimal inconsistent set of clauses that contains a literal $\neg\alpha$ there is a support tree for Δ and α.

Proposition 6. *Let Δ be a clause knowledgebase and let $\Phi \subseteq \Delta$. $\langle \Phi, \alpha \rangle$ is a literal argument iff there is a support tree (N, A, f) for Δ and α such that $\Phi = \{f(x) \mid x \in N\}$.*

Therefore, given a clause knowledgebase Δ and a literal α, we can find all the arguments for α by finding all the subgraphs of the query graph of α in Δ whose clauses can be arranged in a support tree for Δ and α. This helps reduce the computational cost of the process in two ways. First, the search space used when searching for arguments is reduced: instead of an algorithm searching through the whole knowledgebase it can search through the part of the knowledgebase that corresponds to the query graph of α in Δ. Potentially this offers very substantial savings since the query graph may involve a relatively small subset of the formulae in the knowledgebase. Second, the query graph also provides useful information on the attack relation among the clauses its nodes contain. The existence of links among the clauses of the knowledgebase motivates the use of algorithms that follow the paths in the query graph rather than searching through arbitrary subsets of the graph.

The algorithms for searching a query graph are introduced below. The links of the query graph are used to trace paths when searching for arguments and the attack values to which they correspond are used to identify the arcs on the branches of a presupport tree or a support tree.

6 Algorithms

In this section we present the algorithms that can be used to construct and search a query graph in order to find all the literal arguments for α from Δ.

6.1 Algorithm for building the query graph

First we will give a brief description of the $\mathsf{GetFocal}(\Delta, \phi)$ algorithm which retrieves the focal graph of a clause ϕ in a clause knowledgebase Δ, and therefore can be used to retrieve the query graph of α in Δ.

The $\mathsf{GetFocal}(\Delta, \phi)$ algorithm finds the focal graph of ϕ in Δ by doing a depth-first search which follows the links of the component of the attack graph for Δ that is linked to ϕ. Initially all the clauses from Δ are considered as candidates for being clauses in the focal graph of ϕ in Δ and then during the search they can be rejected if they do not satisfy the conditions of the definition of the focal graph. The algorithm chooses the appropriate nodes by using the boolean method $\mathsf{isConnected}(C, \psi)$ which tests whether a clause ψ of the attack graph C is such that each literal $\beta \in \mathsf{Disjuncts}(\psi)$ corresponds to at least one arc to a clause from Δ that has not been rejected . Given the adjacency matrix for the attack graph for Δ, the algorithm locates which clauses of the attack

graph need to be visited. Only those that are linked to ϕ either directly or indirectly with a sequence of arcs from the attack graph for Δ will be visited. From the set of the visited clauses only the ones that satisfy the condition of being connected according to the isConnected function will be clauses in the focal graph.

The algorithm starts by locating clause ϕ in the attack graph. If $\phi \notin \Delta$ or the function $\mathsf{isConnected}(C, \phi)$ returns false, the algorithm returns the empty graph. Otherwise the algorithm, starting from ϕ, follows in a depth-first way all the possible paths through clauses from Δ, indicated by the links of the attack graph and tests whether the isConnected function returns true for the visited nodes. If the function returns false for some clause, then this clause is marked as rejected and the algorithm backtracks to retest whether the rest of the clauses in this path remain connected after this clause has been rejected.

6.2 Algorithm for finding the formulae for each presupport tree (Algorithm 1)

The GetPresupportsTree algorithm constructs a search tree representing an exhaustive search of the query graph of α in Δ in order to find all the different subsets of Δ that can be arranged in a presupport tree structure. Each branch of the search tree is a linked list of nodes which can be accepted or rejected according to the conditions of definition 12. Each of the accepted branches identifies a unique subset of Δ that can be arranged in a presupport tree for Δ and α.

Each node of the search tree denoted $Node$ contains a set of candidates for a presupport tree where each candidate is identified by a clause from the query graph of α in Δ. The set of candidates in a node, denoted $Candidates$, corresponds to a level of a presupport tree for Δ and α. Apart from the value $Candidates$ each $Node$ contains the value $Parent$ as a pointer to its previous $Node$ on the branch, and the value $Ancestors$ which is the set of all the nodes that appear on the same branch above this node.

Each element in $Candidates$ is of the form $Candidate_\phi = (\phi, Attacked_\phi)$ s.t. $\phi \in \Delta$ and $Attacked_\phi \subseteq \mathsf{Disjuncts}(\phi)$ where ϕ represents a potential node of a presupport tree for Δ and α. So each such candidate contains a clause ϕ from the query graph of α in Δ and a subset of $\mathsf{Disjuncts}(\phi)$ denoted $Attacked_\phi$ which keeps track of the disjuncts on which ϕ is attacked by clauses of other candidates from the same branch of the search tree. Each $Candidate_\phi$ is in the $Candidates$ of a $Node$ if there is at least one $Candidate_\psi$ in the $Candidates$ of the parent of the given $Node$ such that ϕ attacks ψ on a disjunct that has not been already attacked by clauses of candidates in the preceding levels (i.e. ancestor nodes).

The root of the search tree, which also represents the root of each of the presupport trees generated by the algorithm, containsas its value for $Candidates$ the set $\{Candidate_{\neg\alpha}\}$. The algorithm then proceeds in a depth-first way in order to construct each branch. Each step of this search corresponds to retrieving all the possible different ϕ s.t. $Candidate_\phi$ can be in the $Candidates$ of a node. A stack is used to store temporarily each $Node$ which will then be replaced by all the possible children nodes for that $Node$. The branch continues being expanded until there is no possible new level, which is either the case when the formulae in the nodes in the branch satisfy all the conditions of being a presupport tree for Δ and α or the case when this set of formulae

Algorithm 1 GetPresupportsTree(Δ, α)

Let S be an empty Stack
Let $QueryGraph = \mathsf{GetFocal}(\Delta \cup \{\neg\alpha\}, \neg\alpha)$
Let $\mathsf{AcceptedBranches} = \emptyset$
Let $rootNode = Node(\{\neg\alpha\}, null)$
push $rootNode$ onto S
while S is not empty **do**
  Let $topNode$ be the top of S
  Let $newNodes = \mathsf{getNewNodes}(QueryGraph, topNode)$
  if $newNodes = \emptyset$ **then**
    if there is $branch \in \mathsf{AcceptedBranches}$ $s.t.$ $branch = \mathsf{getFormulae}(Node)$ **then**
      pop S
    else
      $\mathsf{AcceptedBranches} = \mathsf{AcceptedBranches} \bigcup \{\mathsf{getFormulae}(topNode)\}$
      pop S
    end if
  else
    pop S
    for all $Node \in newNodes$ **do**
      $\mathsf{UpdateAttackValues}(Node)$
      push $Node$ onto S
    end for
  end if
end while
return AcceptedBranches

violates some of the conditions of definition 12. In the first case, when the formulae on the current branch can be arranged in a presupport tree for Δ and α, the formulae of this set excluding $\neg\alpha$ is stored, as long as the same set of formulae has not been stored previously. It is in the last two cases when the algorithm reaches the end of a branch that it moves to the next branch.

In order to control the number of nodes that need to be created, the algorithm is using the subsidiary function $\mathsf{UpdateAttackValues}(Node)$ which updates the value $Attacked_\phi$ of each $Candidate_\phi$ of a newly created $Node$ by testing the attack relation of ϕ with the clauses contained in the rest of the $Candidates$ of its $Ancestors$ nodes.

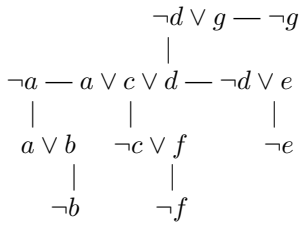
In order to facilitate the search of the query graph of α in Δ denoted $QueryGraph$, the algorithm is using the function $\mathsf{getNewNodes}(QueryGraph, Node)$ which retrieves from the $QueryGraph$ the clauses that attack each candidate of the given node on a disjunct that is not already attacked by candidates of previous nodes and combines them to get all the possible sets of candidates in a way that there is a 1-1 correspondence between the elements of each given candidate and these non-attacked disjuncts. If the $Candidates$ of the updated node contains at least one node $Candidate_\phi$ with $\mathsf{Disjuncts}(\phi) \neq Attacked_\phi$ then the $\mathsf{getNewNodes}$ function returns a non-empty set of all the possible next levels of the given $Node$, otherwise it returns the empty set.

Finally, the $\mathsf{getFormulae}(Node)$ function is used when a leaf node is found and returns the set of formulae on the branch where the given leaf node belongs.

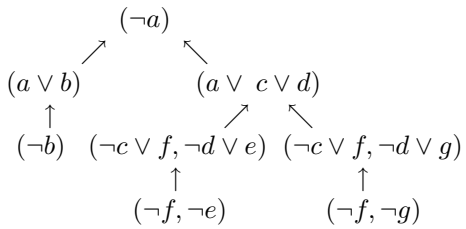
Hence, each of the accepted branches of the algorithm introduced above gives us the set of formulae that can be arranged as a presupport tree for Δ and α. Each node of an accepted branch is selected so as to represent a level of a presupport tree. Each $Candidate_\psi$ in a node's $Candidates$ is such that there is a $Candidate_{\psi'}$ in the $Candidates$ of its parent with $\mathsf{Attacks}(\psi, \psi') \neq null$ and (ψ, ψ') defines an arc of the presupport tree.

The $\mathsf{UpdateAttackValues}(Node)$ algorithm updates each of the candidates of $Node$ according to their attack relation with the candidates of the previous nodes on the same branch in order to ensure that the $\mathsf{getNewNodes}$ algorithm will return a set of children nodes $Children = \{Node_1, \ldots, Node_n\}$ such that for each $Node_i$ from the set $Children$ if $Candidates_i$ is the set of candidates in $Node_i$ and $Candidate_\psi \in Candidates_i$ and $\beta \in \mathsf{Disjuncts}(\psi)$, then there is a no ancestor $Node_a$ of $Node_i$ s.t. $Candidate_{\psi'}$ is in the candidates of $Node_a$ and $\mathsf{Attacks}(\psi, \psi') = \beta$. This ensures that conditions 1) and 3) of the definition of the presupport tree are satisfied. The fact that the candidates of a newly created node cannot be from the candidates that appear on the branch before ensures that condition 2) of the definition of the presupport tree will be satisfied. As a result, all the conditions for an accepted branch to be a presupport tree are met by the algorithm.

Example 19. For $\Delta = \{a \vee b, \neg b, a \vee c \vee d, \neg c \vee f, \neg d \vee e, \neg f, \neg e, \neg d \vee g, \neg g \vee h, c \vee j, \neg k \vee m \vee n, \neg n \vee \neg j, \neg g\}$ and $\alpha = a$, following is the query graph of a in Δ:



The GetPresupportsTree(Δ, α) algorithm generates the following search tree from the above query graph.



Hence, for the branches (numbered from left to right) we have the following sets of formulae:

From branch 1, $\{a \vee b, \neg b\}$
From branch 2, $\{a \vee c \vee d, \neg c \vee f, \neg d \vee e, \neg f, \neg e\}$
From branch 3, $\{a \vee c \vee d, \neg c \vee f, \neg d \vee g, \neg f, \neg g\}$

So each of these sets of formulae can be arranged as a presupport tree.

Since we require arguments for α from Δ, we need to take the output of the algorithm GetPresupportsTree(Δ, α) and determine whether each set of formulae corresponding to a presupport tree can be arranged as a support tree. That is the role of our next algorithm in the next section.

6.3 Algorithm for checking support tree conditions

We now describe the GetSupports algorithm which, using the output of the algorithm presented in section 6.2 (i.e. GetPresupportsTree(Δ, α)), tests whether the set of clauses from each of the accepted branches of the search tree can be arranged as a support tree for Δ and α.

Let $Branches$ denote the output of the GetPresupportsTree(Δ, α). So $Branches$ is a set of sets of formulae. The GetSupports$(Branches)$ algorithm uses the function hasSupport(Γ, α) to test each set $\Gamma \in Branches$ individually. Given a set of clauses Γ, the hasSupport(Γ, α) function generates the presupport trees (N, A, f) where $N = \bigcup\{x \mid f(x) \in \Gamma\}$ and $\neg\alpha$ is the root. The algorithm keeps track of the attack values among the clauses in each presupport tree it generates and these are then used to test whether the additional conditions that differentiate a support tree from a presupport tree are satisfied. When the first such presupport tree that satisfies the conditions of being

minimal and consistent is found, Γ is stored with the set of the accepted supports for literal arguments for α and the next set from $Branches$ is tested. If no such presupport tree exists, then Γ is rejected for being a support for a literal argument for α and the algorithm proceeds by testing with the hasSupport function the next set from $Branches$.

Example 20. Given the results of example 19, if $Branches$ is the output given by the GetPresupportsTree(Δ, α) algorithm, then for each of the sets $\Gamma_1, \Gamma_2, \Gamma_3 \in Branches$ that correspond to to branches 1,2 and 3 respectively, the hasSupport$(\Gamma_i, \alpha), i = 1 \ldots 3$ function returns true and therefore the output of the GetSupports$(Branches)$ algorithm is the set $\Gamma_1, \Gamma_2, \Gamma_3$. Hence, there are three literal arguments for α: $\langle \Gamma_1, \alpha \rangle$, $\langle \Gamma_2, \alpha \rangle$, and $\langle \Gamma_3, \alpha \rangle$.

7 Experimental results

This section covers a preliminary experimental evaluation of the algorithms presented in Section 6 using a prototype implementation programmed in java running on a modest PC (Core2 Duo 1.8GHz).

The experimental data were obtained using randomly generated clause knowledgebases according to the fixed clause length model K-SAT ([22,16]) where the chosen length (i.e. K) for each clause was 3 literals. The 3 disjuncts of each clause were chosen out of a set of N distinct variables (i.e. atoms). Each variable was randomly chosen out of the N available and negated with probability 0.5. For a fixed number of clauses, the number of distinct variables that occur in the disjuncts of all the clauses determines the size of the query graph which in turn determines the size of the search space and hence influences the perfomance of the system. For this reason, 10 different clauses-to-variables ratios were used for each of the different cardinalities tested (where this ratio varied from 1 to 10). For the definition of the ratio we take the integer part of the division of the number of clauses in Δ by the number of variables N (i.e. $\lfloor |\Delta|/|N| \rfloor$).

The evaluation was based on the time consumed by the system when searching for all the literal arguments for a given literal and the randomly generated knowledgebases of 15 to 30 clauses. Hence, for the results presented the smallest number of variables used was 1 and so for the case of a 15 clause knowledgebase, the clauses-to-variables ratio is 10. The largest number of variables used was 30 and so for the case of a 30 clause knowledgebase, and clauses-to-variables ratio is 1.

The preliminary results are presented in Table 1 which contains the median time consumed in milliseconds for 100 repetitions of running the system for each different cardinality and each ratio from 1 to 10. In other words, each field of the table is the median time obtained from finding all the arguments in 100 different knowledgebases of fixed cardinality where the cardinality is determined by the column of the table and the different clauses-to-variables ratios is determined by the row.

From the preliminary results in Table 1, we see that for a low clauses-to-variables ratio (≤ 2) the number of variables is large enough to allow a distribution of the variables amongst the clauses such that it is likely for a literal to occur in a clause without its opposite occurring in another clause from the set. As a result, the query graph tends to contain a small subset of the knowledgebase and the system perfoms relatively quickly.

Table 1. Experimental data

clauses-to-variables ratio	$\|\Delta\| = 15$	$\|\Delta\| = 20$	$\|\Delta\| = 25$	$\|\Delta\| = 30$
1	3.000	6.000	9.000	13.00
2	3.000	6.000	11.00	17.00
3	2.000	6.000	12.50	238.0
4	2.000	5.000	14.00	466.5
5	2.000	4.000	8.000	178.0
6	1.000	3.000	6.500	71.00
7	1.000	5.000	4.000	9.000
8	0.000	1.000	4.000	6.000
9	1.000	1.000	2.000	6.000
10	1.000	2.000	2.000	7.000

The query graph tends also to be small in the case when a relatively small number of variables is distributed amongst the clauses of the knowledgebase (i.e. when the ratio is high) and this makes the occurrence of a variable and its negation in different clauses more frequent. As a result, it is likely for a pair of clauses ϕ, ψ from Δ to be such that $|\mathsf{Preattacks}(\phi, \psi)| > 1$ which will then allow the Attacks relation to be defined among a small number of clauses and therefore the attack graph will involve only a small subset of the knowledgebase. Hence, a large clauses-to-variables ratio also makes the system perform quickly. From these preliminary results the worst case occurs for ratio 4, and this appears to be because the size of the query graph tends to be maximized. This indicates that the clauses-to-variables ratio, rather than the cardinality of the knowledgebase is the dominant factor determining the time perfomance for the system. In future experiments we want to further characterize this worst case perfomance. In particular, we want to better understand the effect of increasing the value for K and so consider clauses with more literals, and we want to better understand the relationship between the number of arguments for a claim that can be obtained from a knowledgebase and the time taken.

8 Discussion

Classical logic has many advantages over defeasible logic for representing and reasoning with knowledge including syntax, proof theory and semantics for the intuitive language incorporating negation, conjunction, disjunction and implication. However, for argumentation, it is computationally challenging to generate arguments from a knowledgebase using classical logic. If we consider the problem as an abduction problem, where we seek the existence of a minimal subset of a set of formulae that implies the consequent, then the problem is in the second level of the polynomial hierarchy [13].

In this paper, we have proposed the use of a connection graph approach as a way of ameliorating the computation cost. The framework we have presented focuses the search for arguments in way that ensures that formulae that have no role as a premise in an argument will not be considered. We have provided theoretical results to ensure the correctness of the proposal, and we have provided provisional empirical results to

indicate the potential advantages of the approach. In furture work, we will extend the empirical evaluation, and extend the theory and algorithms for dealing with arbitrary formulae as claims of arguments.

References

1. L. Amgoud and C. Cayrol. A model of reasoning based on the production of acceptable arguments. *Annals of Mathematics and Artificial Intelligence*, 34:197–216, 2002.
2. S. Benferhat, D. Dubois, and H. Prade. Argumentative inference in uncertain and inconsistent knowledge bases. In *Proceedings of the 9th Annual Conference on Uncertainty in Artificial Intelligence (UAI 1993)*, pages 1449–1445. Morgan Kaufmann, 1993.
3. Ph. Besnard and A. Hunter. A logic-based theory of deductive arguments. *Artificial Intelligence*, 128:203–235, 2001.
4. Ph Besnard and A Hunter. Practical first-order argumentation. In *Proc. of the 20th National Conference on Artificial Intelligence (AAAI'2005)*, pages 590–595. MIT Press, 2005.
5. Ph. Besnard and A. Hunter. *Elements of Argumentation*. MIT Press, 2008.
6. W. Bibel. *Deduction: Automated Logic*. Academic Press, 1993.
7. D. Bryant, P. Krause, and G. Vreeswijk. Argue tuProlog: A lightweight argumentation engine for agent applications. In *Computational Models of Argument (Comma'06)*, pages 27–32. IOS Press, 2006.
8. C. Cayrol, S. Doutre, and J. Mengin. Dialectical proof theories for the credulous preferred semantics of argumentation frameworks. In *Quantitative and Qualitative Approaches to Reasoning with Uncertainty*, volume 2143 of *LNCS*, pages 668–679. Springer, 2001.
9. C. Chesñevar, A. Maguitman, and R. Loui. Logical models of argument. *ACM Computing Surveys*, 32:337–383, 2000.
10. C. Chesñevar, G. Simari, and L. Godo. Computing dialectical trees efficiently in possibilistic defeasible logic programming. In *Proceedings of the 8th Interntational Logic Programming and Non-monotonic Reasoning Conference*, Lecture Notes in Computer Science. Springer, 2005.
11. Y. Dimopoulos, B. Nebel, and F. Toni. On the computational complexity of assumption-based argumentation for default reasoning. *Artificial Intelligence*, 141:57–78, 2002.
12. P. Dung, R. Kowalski, and F. Toni. Dialectical proof procedures for assumption-based admissible argumentation. *Artificial Intelligence*, 170:114–159, 2006.
13. T. Eiter and G. Gottlob. The complexity of logic-based abduction. *Journal of the ACM*, 42:3–42, 1995.
14. M. Elvang-Gøransson, P. Krause, and J. Fox. Dialectic reasoning with classically inconsistent information. In *Proceedings of the 9th Conference on Uncertainty in Artificial Intelligence (UAI 1993)*, pages 114–121. Morgan Kaufmann, 1993.
15. A. García and G. Simari. Defeasible logic programming: An argumentative approach. *Theory and Practice of Logic Programming*, 4(1):95–138, 2004.
16. I. P. Gent and T. Walsh. Easy problems are sometimes hard. *Artificial Intelligence*, 70(1-2):335–345, 1994.
17. A. Kakas and F. Toni. Computing argumentation in logic programming. *Journal of Logic and Computation*, 9:515–562, 1999.
18. R. Kowalski. A proof procedure using connection graphs. *Journal of the ACM*, 22:572–595, 1975.
19. R. Kowalski. *Logic for problem solving*. North-Holland Publishing, 1979.
20. H. Prakken and G. Sartor. Argument-based extended logic programming with defeasible priorities. *Journal of Applied Non-Classical Logics*, 7:25–75, 1997.

21. H. Prakken and G. Vreeswijk. Logical systems for defeasible argumentation. In D. Gabbay, editor, *Handbook of Philosophical Logic*. Kluwer, 2000.
22. B. Selman, D. G. Mitchell, and H. J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1-2):17–29, 1996.
23. G. Vreeswijk. An algorithm to compute minimally grounded and admissible defence sets in argument systems. In *Computational Models of Argument (Comma'06)*, pages 109–120. IOS Press, 2006.