Houdini (unchained): an effective reasoner for defeasible logic

Matteo Cristani[1,†], Guido Governatori[†], Francesco Olivieri[2,†], Luca Pasetto[1,†], Francesco Tubini[1,†], Celeste Veronese[1,†], Alessandro Villa[1,†] and Edoardo Zorzi[1,3,†]

[1]Department of Computer Science, University of Verona, Verona, 37134, Italy
[2]School of Information and Communication Technology, IIIS, Griffith University, Nathan, QLD 4111, Australia
[3]Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland

Abstract

This paper introduces Houdini, a Java implementation of a propositional defeasible logic reasoner. We discuss the development endeavoured so far. Houdini constitutes the first step in a longer plan, we anticipate here as well, that aims at extending the implementation to treat deontic defeasible theories, along with some relevant entities such as amounts, dates and durations. The system is presented in terms of architecture, performance and actual functionalities for the implementation phase documented here, and envisioned in the further steps we are in the process of carrying out in the near future.

Keywords

Defeasible logic; automated reasoning; non-monotonic reasoning.

1. Introduction

We present a novel implementation of propositional defeasible logic, called **Houdini**. It implements a well-known method for computing *extensions* of propositional defeasible theories (namely deductive closures with specific proof tags), based on a forward-chaining propagation algorithm.

The development of Houdini presented here, version 1.0, is the first step of an implementation plan, currently still in progress, which we shall introduce later in Section 6. The implementation language is Java, without any database technology. A reasoner for propositional defeasible logic named SPINdle [1] has already been developed in the past: a comparison in terms of performance between SPINdle and Houdini can be found in Section 4.

The formal definition of the logic is provided in Section 2, while the reasoning algorithms are presented in Section 3, along with the implementation details of the system. Section 5 discusses related work and Section 6 takes some conclusions and sketches further work.

AI[3] *2022, 6th Workshop on Advances in Argumentation in Artificial Intelligence, CEUR Workshop Proceedings (CEUR-WS.org), November 28 - December 02, 2022, Udine, Italy*

[†]These authors contributed equally.

matteo.cristani@univr.it (M. Cristani); guido@governatori.net (G. Governatori); f.olivieri@griffith.edu.au (F. Olivieri); luca.pasetto@univr.it (L. Pasetto); francesco.tubini@univr.it (F. Tubini); celeste.veronese@univr.it (C. Veronese); alessandro.villa@univr.it (A. Villa); edoardo.zorzi@univr.it (E. Zorzi)

2. Defeasible logic

A defeasible theory consists of five different kinds of knowledge: facts, strict rules, defeasible rules, defeaters, and a superiority relation [2].

Let PROP be a set of propositional atoms, Lbl be a set of arbitrary labels. The set $\mathrm{Lit} = \mathrm{PROP} \cup \{\neg p | p \in \mathrm{PROP}\}$ denotes the set of *literals*. The *complement* of a literal q is denoted by $\sim q$; if q is a positive literal p, then $\sim q$ is $\neg p$, and if q is a negative literal $\neg p$ then $\sim q$ is p.

Definition 1. *A* defeasible theory D *is a structure* $(F, R, >)$*, where*

1. $F \subseteq \mathrm{Lit}$ *is a set of facts that denote simple pieces of information that are always considered to be true. For example, a fact is that "Sylvester is a cat", formally* $cat(Sylvester)$*;*
2. R*, the set of rules, contains three types of* rules*: strict rules, defeasible rules, and defeaters.*
3. $> \subseteq R \times R$ *is a binary relation whose transitive closure is acyclic. We refer to this relation as* superiority relaation.

A theory is finite *if the set of facts and rules are finite.*

A *rule* is an expression $r\colon A(r) \hookrightarrow C(r)$ and consists of: (i) A unique name $r \in \mathrm{Lbl}$, (ii) the *antecedent* $A(r)$ which is a finite subset of Lit (also known as the *body* of the rule), (iii) an *arrow* $\hookrightarrow \in \{\rightarrow, \Rightarrow, \rightsquigarrow\}$ denoting, respectively, a strict rule, a defeasible rule and a defeater, and (iv) its *consequent* (or *head*) $C(r) \in \mathrm{Lit}$, which is a single literal. A *strict rule* is a rule in which whenever the premises are indisputable (e.g., facts), then so is the conclusion. For example,

$$cat(X) \rightarrow mammal(X)$$

means that "every cat is a mammal". On the other hand, a *defeasible rule* is a rule that can be defeated by contrary evidence; for example, "cats typically eat birds":

$$cat(X) \Rightarrow eatBirds(X).$$

The underlying idea is that if we know that something is a cat, then we may conclude that it eats birds unless there is evidence proving otherwise. *Defeaters* are rules that cannot be used to draw conclusions directly. Their only use is to prevent some conclusions, i.e., to defeat defeasible rules by producing evidence to the contrary. An example is "if a cat has just fed itself, then it might not eat birds":

$$justFed(X) \rightsquigarrow \neg eatBirds(X).$$

The *superiority relation* $>$ among rules is used to define where one rule may override a second rule for the (opposite) conclusion, e.g., given the defeasible rules

$$\begin{aligned} r\colon cat(X) &\Rightarrow eatBirds(X) \\ r'\colon domesticCat(X) &\Rightarrow \neg eatBirds(X) \end{aligned}$$

which would contradict one another if Sylvester is both a cat and a domestic cat, they do not in fact contradict if we state that r' wins against r, leading to conclude that Sylvester does not to eat birds.

Like in [2], we consider only a propositional version of this logic, and we do not take into account function symbols. Every expression with variables represents the finite set of its variable-free instances.

We use the infix notation $r > s$ to mean that $(r, s) \in >$. The set of strict rules in R is denoted by R_{s}, and the set of strict and defeasible rules by R_{sd}. We name $R[q]$ the set of rules in R whose head is q. A *conclusion* of D is a tagged literal and can have one of the following forms:

- $+\Delta q$, which means that q is definitely provable in D, i.e., there is a definite proof for q, that is a proof using facts, and strict rules only;
- $-\Delta q$, which means that q is definitely not provable, or refuted, in D (i.e., a definite proof for q does not exist);
- $+\partial q$, which means that q is defeasibly provable in D;
- $-\partial q$, which means that q is not defeasibly provable, or refuted, in D.

Given a defeasible theory D, a proof P of length n in D is a finite sequence $P(1), \ldots, P(n)$ of tagged literals of the type $+\Delta q$, $-\Delta q$, $+\partial q$ and $-\partial q$, where the proof conditions defined in the rest of this section hold. $P(1..n)$ denotes the first n steps of proof P.

Given $\# \in \{\Delta, \partial\}$ and a proof P in D, a literal q is *$\#$-provable* in D if there is a line $P(m)$ of P such that $P(m) = +\#q$. A literal q is *$\#$-refuted* in D if there is a line $P(m)$ of P such that $P(m) = -\#q$.

The Definition of Δ describes just forward chaining of strict rules.

$+\Delta$: If $P(n+1) = +\Delta q$ then
(1) $q \in F$ or
(2) $\exists r \in R_s[q] \forall a \in A(r) : +\Delta a \in P(1..n)$.

Literal q is definitely provable if either (1) is a fact, or (2) there is a strict rule for q, whose antecedents have all been definitely proved.

$-\Delta$: If $P(n+1) = -\Delta q$ then
(1) $q \notin F$ and
(2) $\forall r \in R_s[q] \exists a \in A(r) : -\Delta a \in P(1..n)$.

Literal q cannot be definitely proven ($-\Delta q$) if (1) is not a fact and (2) every strict rule for q has at least one definitely refuted antecedent.

The following definition gives the conditions for when a rule is applicable or discarded.

Definition 2. *In the proof condition for $\pm\partial$, a rule $r \in R_{sd}$ is (i)* applicable *iff $\forall a \in A(r)$, $+\partial a \in P(1..n)$; (ii)* discarded *iff $\exists a \in A(r)$ such that $-\partial a \in P(1..n)$.*

We now introduce the proof conditions to show that a literal is defeasibly provable.

$+\partial$: If $P(n+1) = +\partial q$ then
(1) $+\Delta q \in P(1..n)$ or
(2) (2.1) $-\Delta{\sim}q \in P(1..n)$ and
(2.2) $\exists r \in R_{\mathrm{sd}}[q]$ s.t. r is applicable, and
(2.3) $\forall s \in R[{\sim}q]$. either s is discarded, or
(2.3.1) $\exists t \in R[q]$ s.t. t is applicable and $t > s$.

Literal q is defeasibly provable if (1) q is already definitely provable, or (2) we argue using the defeasible part of the theory. For (2), $\sim q$ is not definitely provable (2.1), and there exists an applicable strict or defeasible rule for q (2.2). Every attack s is either discarded (2.3), or defeated by a stronger rule t (2.3.1). When, specifically, no attack exists, namely the literal is supported and there is no support for the opposite literal, then we say that the literal is *irrefutable*.

On the other hand, to prove the a literal is defeasibly refuted ($-\partial$) we have to show that all possible ways to prove it fail. This is encoded by the following proof conditions that correspond to a constructive negation of the conditions for $+\partial$.

$-\partial$: If $P(n+1) = -\partial q$ then
(1) $-\Delta q \in P(1..n)$ and either
(2) (2.1) $+\Delta{\sim}q \in P(1..n)$ or
  (2.2) $\forall r \in R_{\mathrm{sd}}[q]$. either r is discarded, or
  (2.3) $\exists s \in R[\sim q]$ s.t. s is applicable, and
    (2.3.1) $\forall t \in R[q]$. either t is discarded, or $t \not> s$.

Given $\# \in \{\Delta, \partial\}$, a literal p and a theory D, we use $D \vdash \pm\#p$ to denote that there is a proof P in D where for some line i, $P(i) = \pm\#p$. Alternatively, we say that $\pm\#p$ holds in D, or simply $\pm\#p$ holds when the theory is clear from the context. The set of positive and negative conclusions is called *extension*. Formally,

Definition 3. *Given a defeasible theory D, the* extension *of D is defined as $E(D) = (+\Delta, -\Delta, +\partial, -\partial)$, where $\pm\# = \{l : l \text{ appears in } D \text{ and } D \vdash \pm\#l\}$, $\# \in \{\Delta, \partial\}$. We refer to $(+\Delta, -\Delta)$ as the* Definite extension *and $(+\partial, -\partial)$ as the* Defeasible extension.

3. The implementation of Houdini

Houdini is a Spring based web application written in Java. Its architecture consists of three major components: a User Interface, a Parser (which also works as a validator module) and a Reasoner.

As shown in Figure 1, inputs (that is, defeasible logic theories) can be provided to the system through an interactive front-end user interface, either by inserting plain text in the embedded dynamic web-form, or by directly uploading a JSON file (see Figure 2 for a preview of the interface). In the first case, the collection of plain text elements is converted into a JSON structure anyway, in order to give the result to the parser for validation, whereas in the second case the file data is directly fed to the module. Be that as it may, the accepted syntax is the same.

The user can provide three types of information: *facts*, syntactically just single literals (so, any custom alphanumerical string with in addition the character '_' and '~', which can only be prefixed to interpret it as a **negation**), *rules*, specific strings divided into an optional body (comma-separated sequence of literals, or a single literal), an arrow (defining the nature of the rule) and the head (non-optional single literal), and instances of the *superiority relation* (simply two rule names[1] separated by a '>' character, where the left-side one is the superior rule, and the right-side one is instead the inferior rule.)

[1]Rules are automatically labelled with a progressive number, following the inserting order: $r1, r2, \ldots$

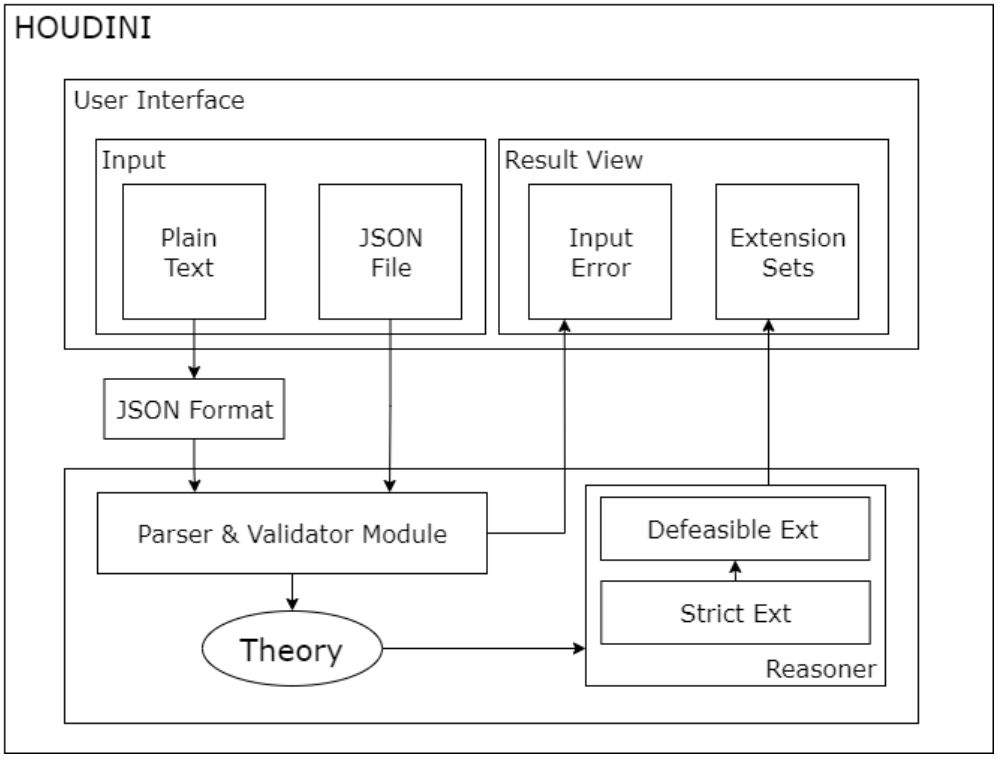


Figure 1: Houdini's architecture and data flow

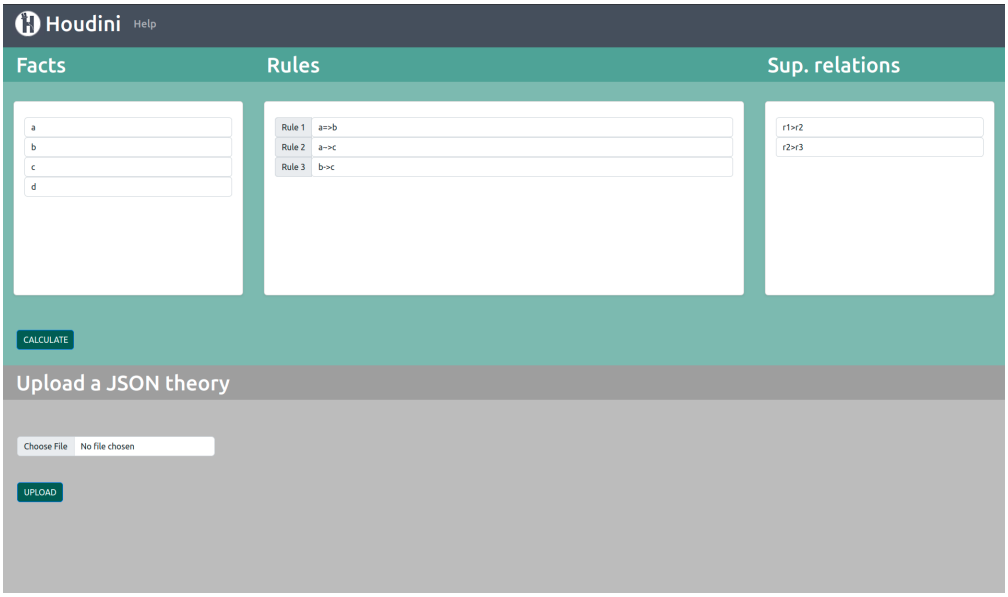


Figure 2: Houdini's front-end offers two ways to provide a theory

After the input is submitted, the parser module obtains the JSON structure and, at first, checks its overall syntactic validity; then, it proceeds to build the object-oriented internal theory representation which will be processed by the reasoner to compute the strict and partial extensions. Algorithm 1 shows the reasoner's workflow.

Algorithm 1 Reasoner

Require: JSONData
1: Theory ← ParserValidator(JSONData)
2: (Theory′, Extension′) ← StrictReasoner(Theory)
3: (Theory″, Extension″) ← DefeasibleReasoner(Theory′, Extension′)
4: **return** Extension″

The user input is processed by an ANTLR standard CFG parser, which validates the data and builds the theory object, later fed to the reasoner modules.

Figure 3 shows the main fields of the Literal and Rule classes. Each literal object contains two sets of rule labels: those referring to rules in which the literal is part of the body, and those referring to rules where the head is the literal. These fields are very useful to guarantee direct access to the rules which are very frequently checked in the reasoning process.

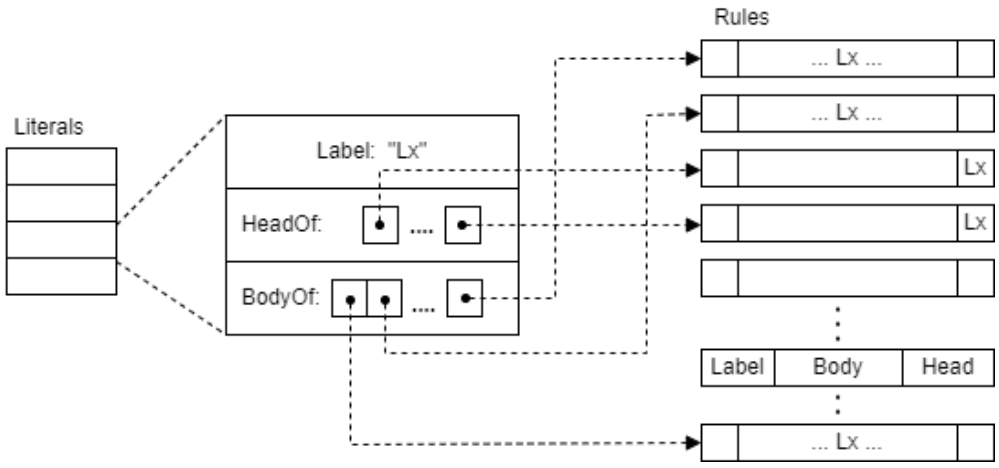


Figure 3: Main attributes of the Literal and Rule classes

The reasoner operates in two modules: Strict Reasoner and Defeasible Reasoner.

Strict Reasoner (Algorithm 2) computes $+\Delta$ and $-\Delta$ and follows an injection strategy to find literals that ought to be added to the definite extension. It starts from the facts, guaranteed to be in $+\Delta$, and removes them from all bodies of rules they're part of. Eventually, new injectable literals are found (heads of completely activated strict rules, i.e. with empty bodies) and the process continues until no new literals are added to the injectable set: when this happens, $+\Delta$ has been completely determined. Then it starts from all those literals that are neither facts nor heads of strict rules (guaranteed to be in $-\Delta$) and starts an analogous process to completely build $-\Delta$. Defeasible Reasoner (Algorithm 3), instead, computes $+\partial$ and $-\partial$. It

starts by restricting the set of the remaining undecided literals and then checks, for each of them, the membership conditions for the two sets. When it finds one of them, it either injects it into the rules or it deactivates it, but, unlike Strict Reasoner, this does not suffice for adding their heads directly to an injectable or deactivation sets. Note that the algorithm loops over the candidates (after having adequately updated them) until the fixed point is reached: this happens when, after an entire loop, no literal has been added to either set; in this case the algorithm ends and returns the conclusions.

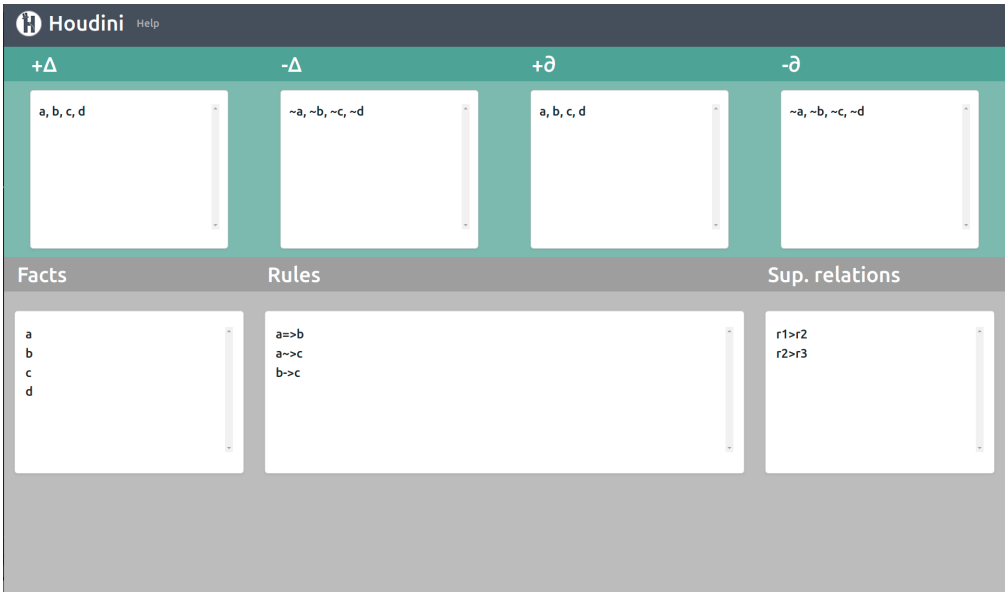


Figure 4: Houdini's front end: computation of the extensions.

4. Performance evaluation

Computing the extension of a propositional defeasible theory can be computed in linear time [3]. After running multiple tests on Houdini's correctness by cross-checking its conclusions results against a test bench of miscellaneous theories encompassing multiple facets of reasoning and corner cases, we test its performances by comparing them with those of SPINdle. SPINdle is a state-of-art implementation of Defeasible Logic that implement different variants of the logic. Recent experiments report that SPINdle outperforms some other implementations of defeasible reasoning [4, 5] and it is comparable to other highly optimised non-monotonic reasoning engines [6]. Furthemore, SPINdle has been successfully used in defeasible logic based applications (in the legal domain) [7, 8]. SPINdle is a Java implementation of defeasible logic that generates the extension of a defeasible theory. The implementation is similar to the implementation presented in this paper with some different choices to prevent performance issues, especially in terms of space occupancy.

Algorithm 2 Strict Extension

Require: Theory ▷ Fed by the validator & parser module

StrictReasoner(Theory)

function StrictReasoner(Theory)
    Extension ← ∅ ▷ All sets ($+\Delta, -\Delta, +\partial, -\partial$) are, at first, empty
    toInject ← $\{l \in \mathcal{L} \mid l \in \mathcal{F}\}$ ▷ All literals that are facts
    toDeactivate ← $\{l \in \mathcal{L} \mid l \notin \mathcal{F} \wedge R_s[l] = \emptyset\}$ ▷ Literals not facts nor heads of strict rules
    for l **in** toInject **do** ▷ If the set is empty it doesn't run
        InjectLiteral(l, toInject, Theory, Extension)
    end for
    for l **in** toDeactivate **do** ▷ If the set is empty it doesn't run
        DeactivateLiteral(l, toDeactivate, Theory, Extension)
    end for
    return Theory, Extension
end function

function InjectLiteral(l, toInject, Theory, Extension)
    for r ∈ $R[l \in \text{body}]$ **do** ▷ All rules with l in their bodies
        Remove l from r.body (update Theory)
        if r.body $= \emptyset$ **then**
            r.tag ← Activated
            if $r \in R_s$ **then** ▷ Must also be a strict rule
                Add r.head to toInject
            end if
        end if
    end for
    Remove l from toInject and add it to $+\Delta$ and $+\partial$ (update Extension)
end function

function DeactivateLiteral(l, toDeactivate, Theory, Extension)
    for r ∈ $R[l \in \text{body}]$ **do**
        r.tag ← StrictlyDeactivated (update Theory)
        if $\forall t \in R_s[r.\text{head}]$ t.tag = StrictlyDeactivated **then** $*$
            Add r.head to toDeactivate
        end if
    end for
    Remove l from toDeactivate and add it to $-\Delta$ (update Extension)
end function

$*$: If, after 'deactivating' a rule r with l in the body we find that, for its head q, this was the last 'not deactivated' rule in $R_s[q]$, add q to the literals to be deactivated

Algorithm 3 Defeasible Extension

Require: Theory, Extension

```
DefeasibleReasoner(Theory, Extension)

function DefeasibleReasoner(Theory, Extension)
    PlusPartialCandidates ← {l ∈ ℒ | l ∉ +Δ ∧ ~l ∈ −Δ}   ▷ Candidates not already in +∂
    MinusPartialCandidates ← {l ∈ ℒ | l ∈ −Δ}
    do
        fixedPoint ← true
        for l in PlusPartialCandidates do
            if ∃r ∈ R_sd[l] s.t. r.tag = Activated then
                isPartial ← true
                for t ∈ R[~l] s.t. t.tag ≠ DefeasiblyDeactivated do
                    versus ← {s ∈ R_sd[l] | s > t}
                    if versus = ∅ ∨ ∄s ∈ versus s.t. s.tag = Activated then
                        isPartial ← false
                        break
                    end if
                end for
                if isPartial = true then
                    injectLiteral(l, Theory, Extension)
                    Remove l from both candidates sets
                    fixedPoint ← false
                end if
            end if
        end for
        for l in MinusPartialCandidates do
            if ∀r ∈ R_sd[l] r.tag = DefeasiblyDeactivated ∨ ~l ∈ +Δ then
                DeactivateLiteral(l, Theory, Extension)
                Remove l from both candidates sets
                fixedPoint ← false
            end if
            isPartial ← false
            for t ∈ R[~l] s.t. t.tag = Activated do
                versus ← {s ∈ R_sd[l] | s > t}
                if versus = ∅ ∨ ∀s ∈ versus, s.tag = DefeasiblyDeactivated then
                    isPartial ← true
                    break
                end if
            end for
            if isPartial = true then
                DeactivateLiteral(l, Theory, Extension)
                Remove l from both candidates sets
                fixedPoint ← false
            end if
        end for
    while fixedPoint = false
    return Theory, Extension
end function
(Continue)
```

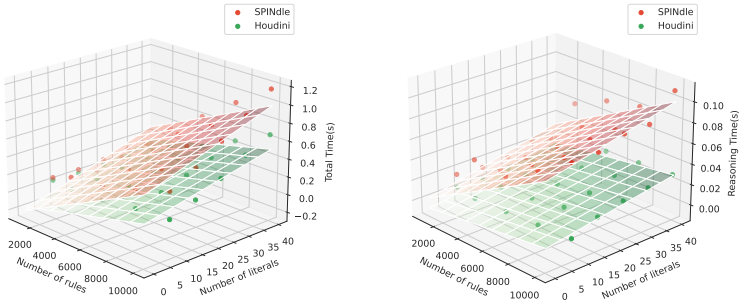
```
(Cont'd)
function InjectLiteral(l, Theory, Extension)
    for r ∈ R[l ∈ body] do                          ▷ All rules with l in their bodies
        Remove l from r.body (update Theory)
        if r.body = ∅ then
            r.tag ← Activated
        end if
    end for
    Add l to +∂ (update Extension)
end function

function DeactivateLiteral(l, Theory, Extension)
    for r ∈ R[l ∈ body] do
        r.tag ← DefeasiblyDeactivated (update Theory)
    end for
    Add l to −∂ (update Extension)
end function
```

We perform two experiments[2] on synthetic theories and compare the two reasoners on two different time statistics: the total time in seconds, i.e., the interval spanning from the start of the loading and parsing phase to the end of the reasoning phase (after which the conclusions are outputted to the user); and the reasoning time in seconds, which includes only the latter.

In the first experiment, we compare the two reasoners across multiple theories varying in size (both the number of rules and the number of literals), whereas in the second experiment we fix the size but we change, randomly, the facts of the theories. These tests have been performed to isolate the performances of Houdini for fixed rule sets, in the perspective of devising a deontic system, where *rules* represent the *normative background* and *facts* the actual events that the law may be governing.

To have meaningful results and, in particular, to assess the performance of the two reasoners in a variety of real-world conditions, the theories are generated randomly and yet always following a certain structure, akin to that of many theories that would stem from user inputs or common use cases. In doing so, we generate a copious amount of tests (which would be difficult with hand-wrought theories) that also vary greatly in their extensions (that is, such that their conclusions sets and $+\partial$, in particular, are not trivial and vary a lot from theory to theory); this is important because it guarantees that the tests are performed on many different situations and not just on a handful of cases. The details on how we proceed with this random generation are specified below.

[2]All tests have been performed locally on a 8GB memory machine, 3.40 Ghz Intel i5, and Ubuntu 18.04 LTS. The Houdini tests have been run in Java 11 whereas the SPINdle ones (version 2.2.4) using Java 8. We completely bypassed any user interface and directly fed theories (in the correct format) to both reasoners, saving, in an external file, the conclusions and the times.



(a) Total time in seconds (b) Reasoning time in seconds

Figure 5: Scatterplot of mean times in seconds (total time and reasoning time only) for SPINdle and Houdini on 25 hyperparameters. The planes are fitted with a normal linear regression on the parameters.

4.1. Experiment 1: different sizes

Given the two parameters n, the number of rules, and w, the 'width', i.e. the number of literals making up the body of any rule (fixed across the whole theory) proceed as follows: starts with w active empty body rules: $\{r_0 \colon \Rightarrow b_0, \ldots, r_{w-1} \colon \Rightarrow b_{w-1}\}$ and then, until the theory comprises of n rules, add r_i whose body is made of w randomly sampled (without replacement) literals taken from $\{b_0, \ldots, b_{i-1}\}$ (i.e. heads of previous rules) and head b_i. With probability pc, set the first literal in the body as X, a literal guaranteed to be defeasibly unprovable, and, independently, with probability ps, add another rule r_{i+1} with a different body and head $\sim b_i$, setting randomly either $r_{i+1} > r_i$ or $r_{i+1} < r_i$ as superiority relation. To limit the sparseness of the theory, after a threshold c the heads will loop over and repeat: for rule $r_i, i > c$ the head will be b_j where $j \equiv i \pmod{c}$.

This procedure generates a random theory that is not completely dissimilar to theories that would derive from real-world situations. In particular, the initial rules are more likely to be activated than later ones, because of a smaller pool of heads to choose from for their bodys, and their activation status depends more on the status of rules that immediately precede them.

The main difference between theories generated this way resides in the randomly inserted unprovable literals X that break 'derivation chains' that, otherwise, would go from the first rule to the last one and would always lead to the same conclusions.

We set $N = \{1000, 2500, 5000, 7500, 10000\}$, $W = \{8, 16, 24, 32, 40\}$, $pc = ps = 0.0025$ and $c = 200$ and generate 20 random theories (as in Section 4.1) for each parameter $(n, w) \in N \times W$. So, in total, we test the two reasoners over 500 theories, 20 theories for 25 combinations of hyperparameters.

For pc and ps percentages are low to avoid theories with too small or large conclusions sets; in particular, we want to avoid both the case where $+\partial$ is only made up of the first w literals $b_0, \ldots, b_{w-1}$ and the case where $+\partial$ includes everything but X. If the probabilities were too high, we would have the first case, if the probabilities were too small, the second one.

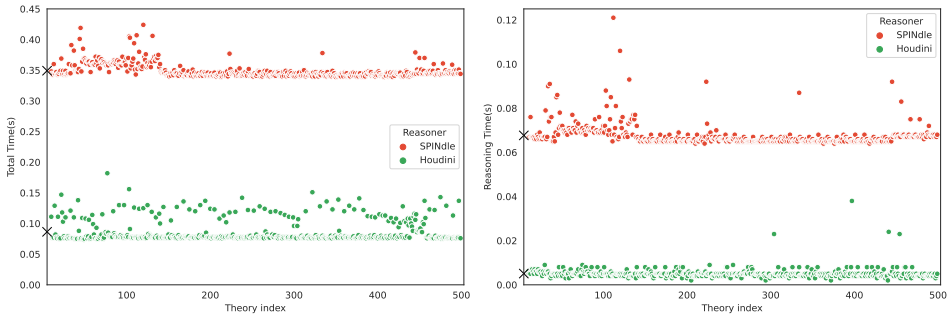
#Rules	Width	Total Time			Reasoning Time		
		Houdini	SPINdle	Improvement	Houdini	SPINdle	Improvement
1000	8	0.0468	0.0763	1.63x	0.0046	0.0174	3.75x
1000	16	0.0307	0.0722	2.35x	0.0038	0.0151	3.94x
1000	24	0.0365	0.0946	2.60x	0.0050	0.0169	3.36x
1000	32	0.0460	0.1247	2.71x	0.0056	0.0247	4.37x
1000	40	0.0579	0.1022	1.76x	0.0096	0.0176	1.84x
2500	8	0.0298	0.1627	5.46x	0.0017	0.0399	22.83x
2500	16	0.0526	0.1516	2.88x	0.0037	0.0204	5.43x
2500	24	0.0842	0.2013	2.39x	0.0060	0.0239	3.96x
2500	32	0.1168	0.2621	2.24x	0.0094	0.0348	3.69x
2500	40	0.1393	0.3402	2.44x	0.0136	0.0609	4.50x
5000	8	0.0605	0.2070	3.42x	0.0030	0.0375	12.31x
5000	16	0.1179	0.2926	2.48x	0.0067	0.0396	5.92x
5000	24	0.1704	0.3989	2.34x	0.0111	0.0483	4.33x
5000	32	0.2321	0.5032	2.17x	0.0177	0.0582	3.29x
5000	40	0.2872	0.6180	2.15x	0.0255	0.0759	2.97x
7500	8	0.1001	0.3163	3.16x	0.0047	0.0565	12.03x
7500	16	0.1762	0.4391	2.49x	0.0120	0.0598	4.97x
7500	24	0.2703	0.5915	2.19x	0.0162	0.0680	4.18x
7500	32	0.3729	0.7387	1.98x	0.0240	0.0688	2.87x
7500	40	0.4680	0.8999	1.92x	0.0273	0.0824	3.02x
10000	8	0.1283	0.4328	3.37x	0.0066	0.0775	11.75x
10000	16	0.2380	0.5854	2.46x	0.0171	0.0780	4.56x
10000	24	0.3735	0.7789	2.09x	0.0226	0.0861	3.81x
10000	32	0.5330	0.9655	1.81x	0.0257	0.0864	3.36x
10000	40	0.6664	1.1684	1.75x	0.0292	0.1097	3.76x

Table 1
Mean times, in seconds, for SPINdle and Houdini with respect to experiment 1.

Results are reported kin Figure 5 and Table 1. Houdini consistently outperforms SPINdle in both statistics. The gains are more sizeable when we consider only the reasoning time (right image, second half of the table): depending on the hyperparameters, we go from just under 2 times faster to up to around 23 times faster; if we do not consider outliers, the improvements are consistently in the 3-4.5x range. For the total time, these somewhat decrease to around 2x: this is consistent with the fact that Houdini has an heavy initialization phase, more so than SPINdle, due to most of the computational improvements and 'tricks' requiring saving and manipulating lots of information before the reasoning phase; moreover, this makes Houdini more memory intensive than SPINdle: we can say that we have found it to be around 1.5x to 2.5x heavier. Improving the parsing strategy, streamlining data structures and devising more sophisticated data management strategies are all avenues that have been considered exploring in the future.

4.2. Experiment 2: different initial hypotheses

The second test suite compares Houdini and SPINdle across theories generated following a slight variation of the random theory generation as in Section 4.1. We call them random theories



(a) Total time in seconds

(b) Reasoning time in seconds

Figure 6: Times in seconds (total time and reasoning time only) for SPINdle and Houdini on 500 random theories with randomly injected hypotheses. In the x-axis the theory index, while in the y-axis the results of that theory. The crosses on the y-axes (note the different scales) mark the mean times.

with random hypotheses. We fix $n = 10000$ and $w = 5$ and do the following: randomly pick t indexes (with no substitution) ranging from 0 to n_i, $\{i'_0, \ldots, i'_{t-1}\}$, and start with rules $r_0 : \Rightarrow b_{i'_0}, \ldots, r_{t-1} : \Rightarrow b_{i'_{t-1}}$; then, from index i_{n_i} onwards proceed before, i.e. generate rules whose bodys comprise previously generated heads until n rules are added. Basically, this has the effect of injecting t random heads, out of n_i, as initial hypotheses.

We generate 500 theories fixing $t = 27$ and $n_i = 30$ so for each theory there are 3 initial heads missing from the injected hypotheses. These numbers have been empirically chosen to limit the sparseness of the theories (the fewer the hypotheses, the more likely it is that the conclusions are trivial, in particular, that $+\partial$ is empty excluding the hypotheses) and to guarantee a good variety of conclusions so that the reasoners may be compared across different situations; also, because of this initial randomness, we set $pc = ps = 0$.

In both scatterplots of Figure 6 we can see how Houdini is faster than SPINdle with and without considering the loading phase. In the first case, Houdini shows a higher variance than SPINdle, but still completely outperforms it: a mean total time of 0.088s versus 0.35s, around 4 times faster. The results considering the reasoning time only are even better: with less variance, Houdini takes an average of 0.005s (5ms) whereas SPINdle 0.068s (68ms), that is, a 13.6x improvement. Of note, execution times are separated: Houdini's worst case is well below SPINdle's best one.

As claimed in Section 4.1, this difference in the results is due to the heavier initialization phase; however, in this case, we perform comparatively better than in the first experiment: the memory penalty has been consistent in the 1.25x to 1.5x range.

5. Related work

A number of investigations have been carried out in the past thirty years regarding algorithms for propositional defeasible logic, starting with the pioneering work of Nute [9] through the technical investigations on algorithmic methods [10] towards extensions to the logical framework that include deontic operators by Nute [11]. As we discussed before SPINdle proved

very successful and superseded previous implementations. After the making of SPINdle a few alternatives/extensions have been proposed with the key focus on large-scale reasoning with instances. [12] proposed a map-reduce parallelisation-based implementation of the algorithm used by SPINdle, while [13] advance a simplified version of the logic to make it more suitable to the parallel implementation. [14] investigates a grounder for SPINdle. However, it is not clear whether such approaches really provide advancements in terms of performance over SPINdle in combination with relational database and related query technology [7, 15].

The most relevant application fields of (deontic) defeasible logic are indeed Legal Reasoning, and Argumentation Analysis. It has been shown that, within the notion of argumentation there is room for a specific semantics of Defeasible Propositional logic in terms of arguments [16].

A related formalism is Defeasible Logic Programming (DeLP) as studied by Leiva et al. [17], that has been implemented by Gàrcia and Simari [18], also in an incremental way by Alfano et al. [19].

6. Conclusions and further work

We have introduced a technology that computes positive (and negative) strict (and defeasible) extensions of a propositional defeasible theory. The technology has been compared to SPINdle, an existing technology. Performances are better, thanks to a careful analysis of drawbacks of SPINdle, overwhelmed in Houdini. The implementation has the following characteristics:

- It provides a web-based interface implemented with a REST/SOAP approach. This can be invoked by a regular web browser, and works on an exposed platform;
- The solution can be accessed also as an API, and it is available on GitHub, in its current version (beta for the moment);
- The solution is effective, as it performs linearly in literals and rules;
- The solution is more performant than SPINdle, being between 2 and 3 times speeder.

The current version of Houdini, 1.0, works on propositional logic. There are some further improvements and features that we are in the process to add. Current implementation plan includes to extend the reasoner to deal with defeasible deontic logic, to add numerical variables such as time and money and to implement support for LegalRuleML-formatted data.

References

[1] H.-P. Lam, G. Governatori, The making of SPINdle, in: G. Governatori, J. Hall, A. Paschke (Eds.), Rule Representation, Interchange and Reasoning on the Web, number 5858 in LNCS, Springer, 2009, pp. 315–322.

[2] G. Antoniou, D. Billington, G. Governatori, M. J. Maher, Representation results for defeasible logic, ACM Transactions on Computational Logic 2 (2001) 255–287.

[3] M. J. Maher, Propositional defeasible logic has linear complexity, Theory and Practice of Logic Programming 1 (2001) 691–711.

[4] S. Batsakis, G. Baryannis, G. Governatori, T. Ilias, G. Antoniou, Legal representation and reasoning in practice: A critical comparison, in: M. Palmirani (Ed.), Jurix 2019, IOS Press, 2018, pp. 31–40.
[5] A. Hecham, M. Croitoru, P. Bisquert, A first order logic benchmark for defeasible reasoning tool profiling, in: C. Benzmüller, F. Ricca, X. Parent, D. Roman (Eds.), Rules and Reasoning, RuleML+RR, LNCS 11092, Springer, 2018, pp. 81–97.
[6] L. Robaldo, S. Batsakis, R. Callegari, F. Calimeri, M. Fujita, G. Governatori, M. C. Morelli, G. Pisano, K. Satoh, I. Tachmazidis, Taking stock of available technologies for compliance checking on first-order knowledge, in: R. Callegari, G. Ciatto, A. Omicini (Eds.), CILC 2022: Italian Conference on Computational Logic, CEUR 3204, 2022.
[7] M. B. Islam, G. Governatori, RuleRS: A rule-based architecture for decision support systems, Artificial Intelligence and Law 26 (2018) 315–344.
[8] G. Governatori, The Regorous approach to process compliance, in: 2015 IEEE 19th International Enterprise Distributed Object Computing Workshop, IEEE Press, 2015, pp. 33–40.
[9] D. Nute, Defeasible logic, in: Handbook of Logic in Artificial Intelligence and Logic Programming, 1994.
[10] G. Antoniou, D. Billington, G. Governatori, M. J. Maher, A. Rock, A family of defeasible reasoning logics and its implementation, in: ECAI 2000, 2000, pp. 459–463.
[11] D. Nute, Norms, priorities, and defeasible logic, in: P. McNamara, H. Prakken (Eds.), Norms, Logics and Information Systems, IOS Press, Amsterdam, 1998, pp. 201–218.
[12] I. Tachmazidis, G. Antoniou, G. Flouris, S. Kotoulas, L. McCluskey, Large-scale parallel stratified defeasible reasoning, in: L. D. Raedt, C. Bessiere, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, P. J. F. Lucas (Eds.), ECAI 2012, volume 242, IOS Press, 2012, pp. 738–743.
[13] M. J. Maher, I. Tachmazidis, G. Antoniou, S. Wade, L. Cheng, Rethinking defeasible reasoning: A scalable approach, Theory and Practice of Logic Programming 20 (2020) 552–586.
[14] M. Rohaninezhad, S. M. Arif, S. Azman Mohd Noah, A grounder for spindle defeasible logic reasoner, Expert Systems with Applications 42 (2015) 7098–7109.
[15] Q. Liu, M. B. Islam, G. Governatori, Towards an efficient rule-based framework for legal reasoning, Knowledge Base Systems 224 (2021) 107082.
[16] G. Governatori, M. J. Maher, D. Billington, G. Antoniou, Argumentation semantics for defeasible logics, Journal of Logic and Computation 14 (2004) 675–702.
[17] M. A. Leiva, A. J. García, P. Shakarian, G. I. Simari, Argumentation-based query answering under uncertainty with application to cybersecurity, Big Data Cogn. Comput. 6 (2022).
[18] A. García, G. Simari, Defeasible logic programming: An argumentative approach, Theory and Practice of Logic Programming 4 (2004) 95–138. doi:10.1017/s1471068403001674.
[19] G. Alfano, S. Greco, F. Parisi, G. Simari, G. Simari, An incremental approach to structured argumentation over dynamic knowledge bases, 2018, pp. 78–87.