REVIEWING JUSTIFICATION-BASED TRUTH MAINTENANCE SYSTEMS FROM A LOGIC PROGRAMMING PERSPECTIVE

HARALD BECK

INFSYS RESEARCH REPORT 17-02
JULY 2017
Abstract. This article reviews Doyle’s classic Justification-based Truth Maintenance System (JTMS) from the perspective of logic programming. More specifically, we use terminology from Answer Set Programming (ASP) to formalize the core JTMS algorithms, which amount to updating an answer set for a program to which a new rule is added. We extend JTMS to analogously account for rule removal. Moreover, we analyze shortcomings of JTMS and the partial correspondence with ASP and give an implementation of the presented algorithms.

Keywords — Truth Maintenance Systems, Answer Set Programming, Knowledge Representation, Nonmonotonic Reasoning, Model Update, Computational Logic.
1 Introduction

Justification-based Truth Maintenance Systems (JTMS) trace back to Doyle’s seminal paper [4] which introduced techniques for maintaining consistent beliefs and their well-foundedness based on justifications. The central concern of JTMS is the incremental model update due to new information which may lead to the retraction of previous conclusions, i.e., nonmonotonic reasoning.

The motivation for this review is that truth maintenance is a classic technique in the area of Knowledge Representation and Reasoning (KR); yet a clear, modular formalization of Doyle’s informal algorithm description seems to be lacking, as well an implementation in a state-of-the-art programming language.

Doyle’s original algorithm is more involved and allows for modelling constraints by means of contradiction nodes. However, the backtracking procedure to resolve them can introduce auxiliary rules which may then lead to models that do not reflect the semantics of the original program. Since our specific interest here is the relationship to Answer Set Programming (ASP), we focus on the core of JTMS that employs no notion of constraints and thus always computes an admissible model, resp. answer set. (For introductions to ASP we refer to [3, 5].) Throughout, we often depart from (resp. add to) Doyle’s original terminology. Used terminology either stems from [2], from logic programming, or was chosen to reflect the conceptual meaning (e.g. algorithm and function names).

2 Truth Maintenance Networks

We start by introducing key notions similarly as [2]. A truth maintenance network (TMN) $\mathcal{T}$ is a pair $(N, J)$, where $N$ is a set of nodes and $J$ is a set of justifications. By a justification we understand an expression of form

$$J = \langle I | O \rightarrow c \rangle,$$

where $I, O \subseteq N$ and $c \in N$. We call $I$ the in-list, $O$ the out-list and $c$ the consequent of $J$. Nodes may be viewed as proposition. The intention of a justification is that the consequent $c$ holds iff all nodes in the in-list hold and no node in the out-list holds. What holds is determined by a model $M \subseteq N$, i.e., a subset of the network’s nodes. Then,

$$\mathcal{J}^M = \{ \langle I | O \rightarrow c \rangle | I \subseteq M \text{ and } O \cap M = \emptyset \},$$

is the set of justifications that are valid in $M$. We say a model $M$ is

(i) **founded**, if there exists a total order $n_1 < \cdots < n_k$ of all elements in $M$ s.t. each $n_j \in M$ has a supporting justification, i.e., some $\langle I | O \rightarrow n_j \rangle \in \mathcal{J}^M$ s.t. $I \subseteq \{ n_1, \ldots, n_{j-1} \}$;

(ii) **closed**, if $n \in M$ for all $\langle I | O \rightarrow n \rangle \in \mathcal{J}^M$; and

(iii) **admissible**, if it is founded and closed.

**Example 1.** As a first example, we illustrate an application of modus ponens. Consider the truth maintenance network $\mathcal{T} = (N, J)$, where $N = \{ x, y \}$ and $J$ consists of the following two justifications.

$$\begin{align*}
(J_1) & \quad \langle \emptyset | \emptyset \rightarrow x \rangle \\
(J_2) & \quad \langle \{ x \} | \emptyset \rightarrow y \rangle
\end{align*}$$

Justification $J_1$ is called a premise (or fact), since both $I$ and $O$ are empty. A premise is valid in every model. Consequently, every model not containing its consequent is not closed. Justification $J_2$ reflects the
material implication $x \supset y$. Assuming we have $x$ in the model, $J_2$ is valid, so $y$ must be concluded. Thus, we get the admissible model $M = \{x, y\}$.

In general, admissible models are not unique, as the next example shows.

**Example 2.** Consider the network $\mathcal{T} = (N, J)$, where $N = \{a, b, c\}$ and $J$ consists of the following three justifications (we omit set braces for $I$ and $O$):

\[
\begin{align*}
(J_1) & \quad \langle b|\emptyset \rightarrow a \rangle \\
(J_2) & \quad \langle \emptyset|c \rightarrow b \rangle \\
(J_3) & \quad \langle \emptyset|a \rightarrow c \rangle
\end{align*}
\]

First consider the empty model. We observe that $J^\emptyset = \{J_2, J_3\}$, since in-lists of these justifications are empty. Consequently, this model is not closed, since $J_2$ would require $b$ and $J_3$ would require $c$ to be concluded. Thus, consider next the model $\{b, c\}$. In this case, the valid justifications are $J^{\{b,c\}} = \{J_1, J_3\}$. The consequent of justification $J_1$, node $a$, is not included in the model, so it is also not closed. Hence, consider model $\{a, b, c\}$, for which we get $J^{\{a,b,c\}} = \{J_1\}$. The absence of $c$ suffices to conclude $b$ via $J_2$, based on which $a$ is founded via $J_1$. Notably, the network has another admissible model: $M_2 = \{c\}$. From the absence of node $a$ alone, $c$ is concluded by $J_3$, which is the only valid justification in $M_2$.

**Logic programming perspective.** Truth maintenance networks resemble normal logic programs in the following way. Let $J = \langle I|O \rightarrow h \rangle$ be a justification such that $I = \{i_1, \ldots, i_n\}$ and $O = \{o_1, \ldots, o_m\}$, and let

\[ r_J = h \leftarrow i_1, \ldots, i_n, \text{not } o_1, \ldots, \text{not } o_m \]  

be the corresponding rule. Moreover, let $P_T = \{r_J \mid J \in J\}$ be the logic program obtained by this translation. The following theorem by Elkan [6] allows for identifying admissible models with answer sets.

**Theorem 1 (cf. [6])** Let $\mathcal{T} = (N, J)$ be a TMN and $M \subseteq N$. Then, (i) $M$ is an admissible model of $\mathcal{T}$ iff it is an answer set of $P_T$. (ii) Deciding whether $\mathcal{T}$ has an admissible model is NP-complete.

As will be discussed below, this correspondence holds only in the absence of odd loops [6], and thus in particular, constraints.

Elkan points out that also incremental reasoning is NP-complete, i.e., given an admissible model $M$ for $P$, deciding for a rule $r$ whether $P \cup \{r\}$ has an admissible model.

**Example 3 (cont’d).** Consider again the network $\mathcal{T} = (N, J)$ of Example 2. We obtain the following translated program $P_T$:

\[
\begin{align*}
(r_1) & \quad a \leftarrow b \\
(r_2) & \quad b \leftarrow \text{not } c \\
(r_3) & \quad c \leftarrow \text{not } a
\end{align*}
\]

This program has two answer sets, $\{a, b\}$ and $\{c\}$, corresponding to the admissible models of $\mathcal{T}$.

In the sequel, we discuss truth maintenance techniques in terms of logic programs:
For a rule of form (1), with atoms \( \{ h, i_1, \ldots, i_n, o_1, \ldots, o_m \} \), \( H(r) = h \) is called the head, \( B^+(r) = \{ i_1, \ldots, i_n \} \) is the positive body, \( B^-(r) = \{ o_1, \ldots, o_m \} \) is the negative body, and
\[
B(r) = B^+(r) \cup B^-(r)
\]
is called the body of \( r \). For a given rule \( r \), we also write \( H, B, B^+, \) and \( B^- \), respectively. We may also denote a rule by \( h \leftarrow B, \) etc.

JTMS Data Structures

By JTMS we will refer to the following data structures based on a program \( P \).

Labels. Each atom \( a \in A_P \) is assigned a unique label. A model corresponds to the set of atoms with label \( in \), all others are (labeled with) \( out \). During the algorithm, a third label \( unknown \) is assigned to atoms that are not yet determined.

Justifications, consequences. The justifications \( J(a) = \{ r \in P \mid H(r) = a \} \) of an atom \( a \) are the rules with head \( a \). Consequences of an atom \( b \) are heads of rules where \( b \) appears in the body, i.e.,
\[
\text{cons}(b) = \{ H(r) \mid r \in P, b \in B(r) \}.
\]

Rule validity. Due to a given labeling label, a rule \( r \) is
- valid, if the body holds, i.e., if \( label(a) = in \) for all \( a \in B^+ \) and \( label(a) = out \) for all \( a \in B^- \);
- invalid if some \( a \in B^+ \) is \( out \) or some \( a \in B^- \) is \( in \); we will call any such atom a spoiler for \( r \);
- posValid if all atoms in \( B^+ \) are \( in \) and no atom in \( B^- \) is \( in \), however, \( B^- \) may contain unknown atoms.

Note that every valid rule is also posValid.

Support. The support of an atom \( a \) specifies the reason why the current label is assigned.
- If \( label(a) = in \), then \( supp(a) \) is the body of some supporting justification, denoted by \( suppJ(a) \), i.e., a valid rule \( r \in P \) with head \( a \). If \( a \) appears as fact, we set \( supp(a) = \emptyset \) since it then needs no further support.
- If \( label(a) = out \), then \( supp(a) \) contains a spoiler for each justification. Notably, the algorithm might fail to find a spoiler for a given rule. Then it can also use an unknown atom \( a \) (from \( B^+ \)) assuming that \( a \) will be labeled \( out \) later. We will explore this below. If \( a \) appears only in rule bodies, then we again set \( supp(a) = \emptyset \).
- If \( label(a) = unknown \), then the conceptual intuition of support does not apply; we technically set \( supp(a) = \emptyset \).

Repercussions. If atom \( a \) supports rule head \( h \), a label change for \( a \) might entail one for \( h \). Thus, \( h \) is said to be affected by \( a \). Formally,
\[
\text{affected}(a) = \{ h \in A_P \mid a \in supp(h) \}.
\]
The repercussions denote the transitive closure of this relation, i.e., the set of atoms which might change their label directly or indirectly due the label change of a given atom.
Algorithm 1: JTMS Algorithm: \textit{add}(r: rule)

\begin{algorithm}
\begin{algorithmic}
\State\textbf{Input:} A rule $r$ with head $h$
\State $\text{register}(r)$
\If{$\text{label}(h) = \text{in}$}: \textbf{return}
\If{$\text{invalid}(r)$}:
\State $\text{supp}(h) := \text{supp}(h) \cup \{\text{spoiler}(r)\}$
\State \textbf{return}
\State $A := \text{repercussions}(h) \cup \{h\}$
\State $\text{update}(A)$
\EndIf
\EndIf
\EndIf
\end{algorithmic}
\caption{JTMS Algorithm: \textit{add}(r: rule)}
\end{algorithm}

Algorithm 2: \textit{register}(r: rule)

\begin{algorithm}
\begin{algorithmic}
\State\textbf{Input:} A rule $r \not\in P$ with atoms $A_r$, head $h$, and body atoms $B$
\ForEach{$a \in A_r \setminus A_P$} \hfill // new atoms
\State $\text{label}(a) := \text{out}$ \hfill // new atoms
\State $\text{cons}(a) := \emptyset$ \hfill // consequences: heads of rules with $a$ in the body
\State $\text{supp}(a) := \emptyset$ \hfill // sufficient atoms supporting the current label of $a$
\State $\text{suppJ}(a) := \text{nil}$ \hfill // in case $a$ is $\text{in}$, a valid rule with head $a$
\State $\text{cons}(b) := \text{cons}(b) \cup \{h\}$
\State $P := P \cup \{r\}$
\EndFor
\end{algorithmic}
\caption{register(r: rule)}
\end{algorithm}

3 The Truth Maintenance Algorithm

Algorithm 1 presents the outline of Doyle’s procedure for adding to a program $P$ with model $M$ a rule $r$. The goal is to update the data structures such that the model $M'$ obtained by atoms with label $\text{in}$ is an answer set of $P \cup \{r\}$.

Example 4. Consider the following rules.

\begin{align*}
r_1 &= a \leftarrow b. \quad r_2 = b \leftarrow \text{not } c. \quad r_3 = a \leftarrow d. \quad r_4 = d \leftarrow c. \\
r_5 &= c \leftarrow d. \quad r_6 = c \leftarrow \text{not } e. \quad r_7 = e \leftarrow .
\end{align*}

Algorithm 1. We start with \textit{add}(r_1). For both atoms $x \in \{a, b\}$, the call $\text{register}(r_1)$ (Algorithm 2) assigns $\text{label}(x) := \text{out}$, $\text{supp}(x) := \emptyset$ and $\text{suppJ}(x) := \text{nil}$. Moreover, $\text{cons}(b) = \{a\}$. Head $a$ is not $\text{in}$ but rule $r_1$ is invalid, so we reassign $\text{supp}(a) := \{b\}$ in Line 4 of Alg. 1 and halt with model $\emptyset$, since no atom has label $\text{in}$. Next, we call \textit{add}(r_2), where $r_2 = b \leftarrow \text{not } c$. Rule head $b$ is not $\text{in}$ but $r_2$ is valid. Due to $r_1$, $a$ is affected by $b$, so \textit{update} (Algorithm 4) is called for $A = \{a, b\}$.

Algorithm 5. After \textit{setUnknown} in Algorithm 4, we try to deterministically assign each atom label $\text{in}$ (resp. $\text{out}$) due to a valid justification (resp. by invalidity of all justifications).

Example 5 (cont’d). Assume \textit{findLabel} is first called for $a$. The justification set is $J(a) = \{r_1\}$, and $r_1$ is neither valid nor invalid. For \textit{findLabel}(b), we have $J(b) = \{r_2\}$ and as $\text{label}(c) = \text{out}$, $r_2$ is valid and
we call setIn cons r stays after adding rules setIn

After we call Example 6 (cont’d).

Algorithm 6 setOut thus assumed to eventually be labeled out x by ensuring that nothing is affected by posValid.
Otherwise, if a leads to 6 INFSYS RR 17-02

Algorithm 3: spoiler(r : rule)

| Input: A rule r ∈ P with atoms Ar, pos/neg. body B+/B− |
| Output: If r is invalid, an atom a ∈ Ar supporting the invalidity, else nil |

1 if randomBoolean() 2 if ∃a ∈ B+: label(a) = out return a 3 else if ∃a ∈ B−: label(a) = in return a 4 else return nil 5 else 6 if ∃a ∈ B−: label(a) = in return a 7 else if ∃a ∈ B+: label(a) = out return a 8 else return nil

Algorithm 4: update(A: set of atoms)

| foreach a ∈ A: setUnknown(a) |
| foreach a ∈ A: findLabel(a) // try to determine first |
| foreach a ∈ A: chooseLabel(a) // else make choices |

we call setIn(r2). This assigns label(b) := in, supp(b) := {e} and suppJ(b) := r2. As labelSet is true, cons(b) = {a} and a is unknown, we recursively call findLabel(a) for propagation; now r1 is valid. In setIn(r1) we set label(a) := in, supp(a) := {b} and suppJ(a) := r1. We have the model {a, b} which stays after adding rules r3, r4 and r5; adding r6 leads to {a, c, d}. We now call add(r7), where r7 = e ←. After findLabel, only e is assigned (in), and a, b, c, d remain unknown.

Algorithm 6. Procedure chooseLabel will similarly assign label(x) := out if all justifications are not posValid. Otherwise, if a posValid justification is found for atom x we check whether assigning label(x) := in is safe by ensuring that nothing is affected by x so far. This may happen only if x is already used as a spoiler and thus assumed to eventually be labeled out. Then, Lines 14-16 reset x and affected atoms to unknown and recursively calls the subprocedure.

Example 6 (cont’d). We call chooseLabel(d) and r4 ∈ J(d) is not posValid (c is unknown). We call setOut(d) and a, c are unknown consequences of d; chooseLabel(a) leads to setOut(a) and chooseLabel(c) leads to setOut(c). Now b is an unknown consequence of c and chooseLabel(b) finds posValid(r2). However, the previous call setOut(a) set supp(a) := {b, d}, thus affected(b) = {a} ≠ ∅. Hence, we call setUnknown for b, a. We enter chooseLabel(b), then setIn(b); finally chooseLabel(a), then setIn(a) and get model {a, b, e}.

4 Extending JTMS: Removing Rules

Doyle provided no explicit means to remove rules. However, we can simulate removal: we can store a rule r = h ← B internally as h ← B, not rmB, where rmB is a fresh atom. Then, removing r amounts to adding the fact rmB ←. However, reactivating the rule can then only be done by adding a modified copy of form h ← B, not rmB′, where rmB′ is another fresh atom. Consequently, this approach will lead to an inflation of the knowledge base.
Algorithm 5: findLabel(a: atom)

1 if label(a) ≠ unknown: return
2 labelSet := false
3 J(a) := \{r ∈ P | H(r) = a\} // H(r) is the head of r
4 if ∃r ∈ J(a): valid(r)
5     setIn(r)
6     labelSet := true
7 else if ∀r ∈ J(a): invalid(r)
8     setOut(a)
9     labelSet := true
10 if labelSet
11     foreach u ∈ unknownCons(a): findLabel(u)

12 defn unknownCons(a) = \{c ∈ cons(a) | label(c) = unknown\}
13 def setIn(r):
14     h := H(r)
15     label(h) := in
16     supp(h) := B(r)
17     suppJ(h) := \{r\}
18 def setOut(a):
19     label(a) := out
20     supp(a) := \{\text{spoiler}(r) | r ∈ P and H(r) = a\}
21     suppJ(a) := nil
22 def setUnknown(a):
23     label(a) := unknown
24     supp(a) := ∅
25     suppJ(a) := nil

A better solution is presented in Algorithm 7 which allows for removing a rule in analogy to Algorithm 1. First, we potentially remove in deregister (Algorithm 8) obsolete entries in the TMS data structures. This involves removal of atoms that are no longer in the program and updating consequences of the remaining ones. Procedure remove then halts without updates if rule head h no longer occurs in the program, h has label out, or if r was not the supporting justification of h.

Example 7 (cont’d). Suppose we delete \(r_3 = a ← d\), i.e., we call remove\(r_3\). Both a and d remain in the program, so in deregister\(r_3\) we only remove a from cons\(d\). We then halt in Line 3 (of Algorithm 7) since supp\(J(a) = r_1\). Consequently, the model \(\{a, b, e\}\) is maintained.

If no exit criterion applies, we determine repercussions as in Algorithm 1 (add) and call the same update procedure.
Algorithm 6: chooseLabel(a: atom)

1 if label(a) ≠ unknown: return
2 labelSet := false
3 J(a) := \{r ∈ P | H(r) = a\}
4 if ∃ r ∈ J(a): posValid(r)
5 if affected(a) = ∅
6 setIn(r)
7 labelSet := true
8 else
9 setOut(a) // for support: view unknown as out
10 labelSet := true
11 if labelSet
12 foreach u ∈ unknownCons(a): chooseLabel(u)
13 else
14 C := affected(a) ∪ \{a\}
15 foreach c ∈ C: setUnknown(c)
16 foreach c ∈ C: chooseLabel(c)

Algorithm 7: JTMS Extension: remove(r: rule with head h)

1 deregister(r)
2 if h /∈ AP\{r\} or label(h) = out or suppJ(h) /≠ r
3 return
4 A := repercussions(h) ∪ \{h\}
5 update(A)

Algorithm 8: deregister(r)

Input: A rule r with atoms Ar, head h, and body atoms Br

1 if r /∈ P return
2 P := P \{r\}
3 foreach a ∈ Ar \ AP // deprecated rule atoms
4 remove key a in maps label, cons, supp, suppJ
5 foreach b ∈ Br ∩ AP // body atoms that are still in use
6 if \{r' ∈ P | H(r') = h and b ∈ Br'\} = ∅
7 cons(b) := cons(b) \ {h}
5 Analysis of JTMS

Doyle’s algorithm faces some problem on its own, and some with respect to the (partial) correspondence with Answer Set Programming.

Algorithm design. In [4], Doyle described his algorithm only informally. Our formal presentation uses new vocabulary and is modularized into subprocedures. A design issue is the 2-step approach to deterministic and non-deterministic label assignment in Algorithm 4. Doyle points out that \texttt{findLabel} (in our terms) is subsumed by \texttt{chooseLabel} and only introduced for efficiency. However, from this perspective, it is unclear why in Algorithm 6, Line 4, one would not first look for \texttt{valid} rules, then for \texttt{posValid} ones, etc. To first compute the deterministic effects after a single choice is natural, as this reduces the probability that retraction steps of Lines 14-16 will be needed.

Example 8. In Example 6, assume a deterministic step after the initial call \texttt{chooseLabel(d)}, with the same order as before: By contrast, \texttt{a} is now examined in \texttt{findLabel} and remains \texttt{unknown}. Next, \texttt{findLabel(c)} will call \texttt{setOut(c)}, leading label \texttt{in} for \texttt{b} and then \texttt{a} due to recursive calls of \texttt{findLabel}.

Odd loops. As pointed out in [7], the update may lead to inadmissible models. JTMS cannot handle odd loops, i.e., an odd number of negations due to which an atom might mistakenly create a cyclic foundation for itself. The minimal case is shown in the next example.

Example 9. Consider the rule \( r = x \leftarrow \lnot x \), where \( x \) is a fresh atom; \( r \) is valid and \texttt{findLabel(x)} calls \texttt{setIn(r1)} which assigns \( \text{supp}(x) := \{x\} \), i.e., \( x \) supports itself. We get the inadmissible model \( \{x\} \) instead of none.

Note that for odd loops as in Example 9, simple ad-hoc tests (e.g. for self-support) would suffice. In general, however, odd loops cannot be solved that easily and require a different kind of maintenance technique.

Odd loops may not only result in inadmissible models, they may also cause non-termination.

Example 10. Consider an empty program, then \( \text{add}(r1) \) for \( r1 = a \leftarrow \lnot b \) and \( \text{add}(r2) \) for \( r2 = b \leftarrow a \), which is valid. Since \( b \) is \texttt{out}, we call \texttt{update\{a,b\}}; atom \( a \) will be assigned \texttt{in} since \( r1 \) is \texttt{posValid}; \( \text{supp}(a) := \{b\} \). For atom \( b \) we have \texttt{posValid(r2)} but \texttt{affected(b)} \( \neq \emptyset \). Thus, \( a \) and \( b \) are reset to \texttt{unknown}, which also occurs when starting with \( b \).

Constraints and inconsistency. More abstractly, JTMS cannot deal with inconsistency. In particular, a constraint \( \bot \leftarrow i1, \ldots, in, \lnot o1, \ldots, \lnot om \) cannot be modelled by

\[
x \leftarrow i_1, \ldots, i_n, \lnot o_1, \ldots, \lnot o_m, \lnot x,
\]

as usual as this rule introduces a (minimal) odd loop. Doyle presented a workaround: when a special contradiction node (atom) \( c \) is derived, a dependency-directed backtracking procedure will add new rules \( R \) to the program \( P \) to prevent the derivation of \( c \). However, the admissible models of \( P \cup R \) are not necessarily admissible for \( P \). We thus skip a discussion of this procedure.

Non-termination. Another issue concerns the unsystematic backtracking in Algorithm 6, where the recursive call in Line 16 may lead to an infinite loop.
Example 11. Consider again the program from Example 4 and let the truth maintenance network be in the same state (in terms of contents of all data structures) as in Example 6 where \( b \) is the last remaining unknown atom. Let us call this specific state \( U_b \). There, we found \( \text{posValid}(r_2) \) (Line 4), but \( \text{affected}(b) = \{a\} \) (Line 5), hence we reset \( a \) and \( b \) to unknown (Line 15). Let this state be denoted by \( U_{ab} \). In Example 6 we assumed that in Line 16 we will first pick atom \( b \). Let us now assume in state \( U_{ab} \), the implementation first picks \( a \), i.e., the call \( \text{chooseLabel}(a) \). Now, neither of the justifications \( r_1 = a \leftarrow b \) nor \( r_3 = a \leftarrow d \) are \( \text{posValid} \) (label(b) = unknown, label(d) = out). Consequently, \( a \) is set to out and we again enter state \( U_b \), leading to state \( U_{ab} \), and so forth.

In practice, such infinite loops can be avoided by shuffling the order of atoms for which the recursive call to \( \text{chooseLabel} \) is made.

Example 12 (cont’d). Assume set \( C \) of Line 14 is represented as list and randomly shuffled after initialization. Then, eventually we will first call \( \text{chooseLabel}(b) \) in state \( U_{ab} \) as in Example 6, where the algorithm terminates with the correct model \( \{a, b, e\} \).

6 Implementation & Application

We provide a clean implementation of the JTMS algorithms as presented. The source code, written in Scala, is available here:

- https://github.com/hbeck/jtms

This realization focuses on conceptual clarity and follows closely the pseudo code of this paper. The main difference is that truth maintenance network data structures such as \( \text{cons}, \text{supp} \) and \( \text{label} \), as well as network specific functions such as \( \text{valid}, \text{invalid} \) and \( \text{posValid} \) have been factored out in a separate class \( \text{SimpleNetwork} \) which derives from the trait \( \text{TruthMaintenanceNetwork} \).

A more efficient variant can be found in the class \( \text{OptimizedNetwork} \) in the source code of Ticker, available at https://github.com/hbeck/ticker. Ticker is an ASP-based stream reasoning engine that utilizes JTMS for incremental model update as described. We refer the interested reader to [1]. The optimized version essentially avoids frequent evaluations by maintaining caches. However, the algorithm itself remains the same.

7 Conclusion

Doyle’s Justification-based Truth Maintenance System is a classic nonmonotonic reasoning technique. We presented a formalization of its core in logic programming terms and gave an extension for rule removal. Moreover, we briefly analyzed algorithmic and semantic shortcomings of JTMS, in particular with respect to correspondence with Answer Set Programming. Finally, we gave implementations of the presented algorithms.

Acknowledgements

Christian Folie was involved in the development of both JTMS implementations. In particular, he contributed substantially to the Ticker project.
References


