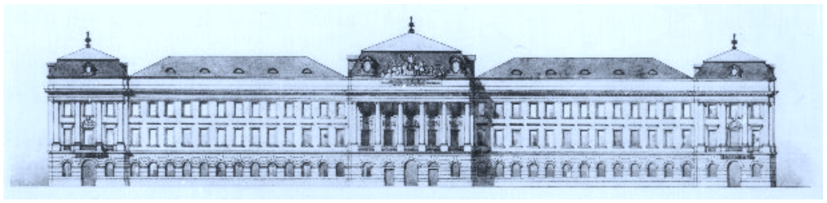
INFSYS

RESEARCH

REPORT



INSTITUT FÜR INFORMATIONSSYSTEME

ARBEITSBEREICH WISSENSBASIERTE SYSTEME

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

HARALD BECK

INFSYS RESEARCH REPORT 17-02

JULY 2017

Institut für Informationssysteme
AB Wissensbasierte Systeme
Technische Universität Wien
Favoritenstraße 9-11
A-1040 Wien, Austria
Tel: +43-1-58801-18405
Fax: +43-1-58801-18493
sek@kr.tuwien.ac.at
www.kr.tuwien.ac.at

INFSYS RESEARCH REPORT

INFSYS RESEARCH REPORT 17-02, JULY 2017

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

Harald Beck[1]

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]Institut für Informationssysteme, Technische Universität Wien, Favoritenstraße 9-11, A-1040 Vienna, Austria; email: beck@kr.tuwien.ac.at.

Acknowledgements: This research has been supported by the Austrian Science Fund (FWF) projects P26471 and W1255-N23.

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 $(N, \mathcal{J})$, where N is a set of *nodes* and $\mathcal{J}$ is a set of *justifications*. By a *justification* we understand an expression of form

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

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,

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

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 $\langle I | O \to n_j \rangle \in \mathcal{J}^M$ s.t. $I \subseteq \{n_1, \ldots, n_{j-1}\}$;

(ii) *closed*, if $n \in M$ for all $\langle I | O \to n \rangle \in \mathcal{J}^M$; 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 $\mathcal{T} = (N, \mathcal{J})$, where $N = \{x, y\}$ and $\mathcal{J}$ consists of the following two justifications.

$$\begin{array}{ll} (J_1) & \langle \emptyset | \emptyset \to x \rangle \\ (J_2) & \langle \{x\} | \emptyset \to y \rangle \end{array}$$

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

$$\begin{array}{ll} (J_1) & \langle b|\emptyset \rightarrow a\rangle \\ (J_2) & \langle \emptyset|c \rightarrow b\rangle \\ (J_3) & \langle \emptyset|a \rightarrow c\rangle \end{array}$$

First consider the empty model. We observe that $\mathcal{J}^{\emptyset} = \{J_2, J_3\}$, 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 $\mathcal{J}^{\{b,c\}} = \{J_1, J_3\}$. 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 $\mathcal{J}^{\{a,b,c\}} = \{J_1\}$. In other words, this model is not founded: there are no valid justifications for nodes b and c, and J_1 is also not a supporting justification for a.

By removing c from the model, we get the admissible model $M_1 = \{a, b\}$, for which both J_1 and J_2 are valid. 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 $I = \{i_1, \dots, i_n\}$ and $O = \{o_1, \dots, o_m\}$, and let

$$r_J \;=\; h \leftarrow i_1, \dots, i_n, \mathrm{not}\, o_1, \dots, \mathrm{not}\, o_m \tag{1}$$

be the corresponding rule. Moreover, let $P_{\mathcal{T}} = \{r_J \mid J \in \mathcal{J}\}$ 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 $\mathcal{T} = (N, \mathcal{J})$ be a TMN and $M \subseteq N$. Then, (i) M is an admissible model of $\mathcal{T}$ iff it is an answer set of $P_{\mathcal{T}}$. (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 $\mathcal{T} = (N, \mathcal{J})$ of Example 2. We obtain the following translated program $P_{\mathcal{T}}$:

$$\begin{array}{lcl} (r_1) & a \leftarrow & b \\ (r_2) & b \leftarrow & \mathrm{not}\, c \\ (r_3) & c \leftarrow & \mathrm{not}\, a \end{array}$$

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:

- a program P (i.e. a set of rules) replaces justifications, and
- atoms A_P occurring in P replace nodes.

For a rule of form (1), with atoms $\{h, i_1, \dots, i_n, o_1, \dots, o_m\}$, $H(r) = h$ is called the *head*, $B^+(r) = \{i_1, \dots, i_n\}$ is the *positive body*, $B^-(r) = \{o_1, \dots, o_m\}$ 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.,

$$cons(b) = \{H(r) \mid r \in P, b \in B(r)\}.$$

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,

$$affected(a) = \{h \in A_P \mid a \in supp(h)\}.$$

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: $add(r\colon rule)$
  Input: A rule $r$ with head $h$
1 $register(r)$
2 if $label(h) = in$: return
3 if $invalid(r)$:
4     $supp(h) := supp(h) \cup \{spoiler(r)\}$
5     return
6 $A := repercussions(h) \cup \{h\}$
7 $update(A)$
```

```
Algorithm 2: $register(r\colon rule)$
  Input: A rule $r \notin P$ with atoms $A_r$, head $h$, and body atoms $B$
1 foreach $a \in A_r \setminus A_P$    // new atoms
2     $label(a) := out$
3     $cons(a) := \emptyset$        // consequences: heads of rules with $a$ in the body
4     $supp(a) := \emptyset$        // sufficient atoms supporting the current label of $a$
5     $suppJ(a) := nil$    // in case $a$ is $in$, a valid rule with head $a$
6 foreach $b \in B$
7     $cons(b) := cons(b) \cup \{h\}$
8 $P := P \cup \{r\}$
```

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 in is an answer set of $P \cup \{r\}$.

Example 4. Consider the following rules.

$$r_1 = a \leftarrow b.\ \ r_2 = b \leftarrow \text{not}\, c.\ \ r_3 = a \leftarrow d.\ \ r_4 = d \leftarrow c.$$
$$r_5 = c \leftarrow d.\ \ r_6 = c \leftarrow \text{not}\, e.\ \ r_7 = e \leftarrow .$$

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

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

Example 5 (cont'd). Assume *findLabel* is first called for a. The justification set is $J(a) = \{r_1\}$, and r_1 is neither valid nor invalid. For *findLabel*(b), we have $J(b) = \{r_2\}$ and as $label(c) = out$, r_2 is valid and

Algorithm 3: $spoiler(r\colon$ *rule*$)$

```
Input: A rule $r \in P$ with atoms $A_r$, pos./neg. body $B^+/B^-$
Output: If $r$ is invalid, an atom $a \in A_r$ supporting the invalidity, else $nil$
1 if $randomBoolean()$
2     if $\exists\, a \in B^+\colon label(a) = out$ return $a$
3     else if $\exists\, a \in B^-\colon label(a) = in$ return $a$
4     else return $nil$
5 else
6     if $\exists\, a \in B^-\colon label(a) = in$ return $a$
7     else if $\exists\, a \in B^+\colon label(a) = out$ return $a$
8     else return $nil$
```

Algorithm 4: $update(A\colon$ *set of atoms*$)$

```
1 foreach $a \in A\colon$ $setUnknown(a)$
2 foreach $a \in A\colon$ $findLabel(a)$    // try to determine first
3 foreach $a \in A\colon$ $chooseLabel(a)$    // else make choices
```

we call $setIn(r_2)$. This assigns $label(b) := in$, $supp(b) := \{c\}$ and $suppJ(b) := r_2$. As $labelSet$ is true, $cons(b) = \{a\}$ and a is $unknown$, we recursively call $findLabel(a)$ for propagation; now r_1 is valid. In $setIn(r_1)$ we set $label(a) := in$, $supp(a) := \{b\}$ and $suppJ(a) := r_1$. We have the model $\{a, b\}$ which stays after adding rules r_3, r_4 and r_5; adding r_6 leads to $\{a, c, d\}$. We now call $add(r_7)$, where $r_7 = e \leftarrow$. 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 $r_4 \in J(d)$ 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 $posValid(r_2)$. However, the previous call $setOut(a)$ set $supp(a) := \{b, d\}$, thus $affected(b) = \{a\} \neq \emptyset$. 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 $r = h \leftarrow B$ internally as $h \leftarrow B, \text{not } rm_r$, where rm_r is a fresh atom. Then, removing r amounts to adding the fact $rm_r \leftarrow$. However, reactivating the rule can then only be done by adding a modified copy of form $h \leftarrow B, \text{not } rm'_r$, where rm'_r is another fresh atom. Consequently, this approach will lead to an inflation of the knowledge base.

Algorithm 5: *findLabel*$(a\colon atom)$

```
if $label(a) \neq unknown$: return
$labelSet := false$
$J(a) := \{r \in P \mid H(r) = a\}$    // $H(r)$ is the head of $r$
if $\exists\, r \in J(a)$: $valid(r)$
    $setIn(r)$
    $labelSet := true$
else if $\forall\, r \in J(a)$: $invalid(r)$
    $setOut(a)$
    $labelSet := true$
if $labelSet$
    foreach $u \in unknownCons(a)$: $findLabel(u)$
--------------------------------------------------------------
defn $unknownCons(a) = \{c \in cons(a) \mid label(c) = unknown\}$
def $setIn(r)$:
    $h := H(r)$
    $label(h) := in$
    $supp(h) := B(r)$
    $suppJ(h) := r$
def $setOut(a)$:
    $label(a) := out$
    $supp(a) := \{\, spoiler(r) \mid r \in P$ and $H(r) = a \,\}$
    $suppJ(a) := nil$
def $setUnknown(a)$:
    $label(a) := unknown$
    $supp(a) := \emptyset$
    $suppJ(a) := nil$
```

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 $r_3 = a \leftarrow d$, 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 $suppJ(a) = r_1$. 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: $\mathit{chooseLabel}(a\colon \textit{atom})$

```
1  if $\mathit{label}(a) \neq \mathit{unknown}$:  return
2  $\mathit{labelSet} := \mathit{false}$
3  $J(a) := \{r \in P \mid H(r) = a\}$
4  if $\exists\, r \in J(a)\colon \mathit{posValid}(r)$
5      if $\mathit{affected}(a) = \emptyset$
6          $\mathit{setIn}(r)$
7          $\mathit{labelSet} := \mathit{true}$
8  else
9      $\mathit{setOut}(a)$        // for support: view $\mathit{unknown}$ as $\mathit{out}$
10     $\mathit{labelSet} := \mathit{true}$
11 if $\mathit{labelSet}$
12     foreach $u \in \mathit{unknownCons}(a)$:  $\mathit{chooseLabel}(u)$
13 else
14     $C := \mathit{affected}(a) \cup \{a\}$
15     foreach $c \in C$:  $\mathit{setUnknown}(c)$
16     foreach $c \in C$:  $\mathit{chooseLabel}(c)$
```

Algorithm 7: JTMS Extension: $\mathit{remove}(r\colon \textit{rule with head } h)$

```
1  $\mathit{deregister}(r)$
2  if $h \notin A_{P\setminus\{r\}}$ or $\mathit{label}(h) = \mathit{out}$ or $\mathit{suppJ}(h) \neq r$
3      return
4  $A := \mathit{repercussions}(h) \cup \{h\}$
5  $\mathit{update}(A)$
```

Algorithm 8: $\mathit{deregister}(r)$

```
   Input: A rule $r$ with atoms $A_r$, head $h$, and body atoms $B(r)$
1  if $r \notin P$ return
2  $P := P \setminus \{r\}$
3  foreach $a \in A_r \setminus A_P$    // deprecated rule atoms
4      remove key $a$ in maps $\mathit{label}$, $\mathit{cons}$, $\mathit{supp}$, $\mathit{suppJ}$
5  foreach $b \in B(r) \cap A_P$    // body atoms that are still in use
6      if $\{r' \in P \mid H(r') = h \textit{ and } b \in B(r')\} = \emptyset$
7          $\mathit{cons}(b) := \mathit{cons}(b) \setminus \{h\}$
```

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 *findLabel* (in our terms) is subsumed by $\mathit{chooseLabel}$ and only introduced for efficiency. However, from this perspective, it is unclear why in Algorithm 6, Line 4, one would not first look for *valid* rules, then for $\mathit{posValid}$ 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 $\mathit{chooseLabel}(d)$, with the same order as before: By contrast, a is now examined in *findLabel* and remains $\mathit{unknown}$. Next, $\mathit{findLabel}(c)$ will call $\mathit{setOut}(c)$, leading label in for b and then a due to recursive calls of $\mathit{findLabel}$.

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 $r = x \leftarrow \mathrm{not}\, x$, where x is a fresh atom; r is valid and $\mathit{findLabel}(x)$ calls $\mathit{setIn}(r_1)$ which assigns $\mathit{supp}(x) := \{x\}$, 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 $\mathit{add}(r_1)$ for $r_1 = a \leftarrow \mathrm{not}\, b$ and $\mathit{add}(r_2)$ for $r_2 = b \leftarrow a$, which is valid. Since b is out, we call $\mathit{update}(\{a, b\})$; atom a will be assigned in since r_1 is $\mathit{posValid}$; $\mathit{supp}(a) := \{b\}$. For atom b we have $\mathit{posValid}(r_2)$ but $\mathit{affected}(b) \neq \emptyset$. Thus, a and b are reset to $\mathit{unknown}$, which also occurs when starting with b.

Constraints and inconsistency. More abstractly, JTMS cannot deal with inconsistency. In particular, a constraint $\bot \leftarrow i_1, \ldots, i_n, \mathrm{not}\, o_1, \ldots, \mathrm{not}\, o_m$ cannot be modelled by

$$x \leftarrow i_1, \ldots, i_n, \mathrm{not}\, o_1, \ldots, \mathrm{not}\, o_m, \mathrm{not}\, x\,, \tag{2}$$

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 $posValid(r_2)$ (Line 4), but $affected(b) = \{a\}$ (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 $chooseLabel(a)$. Now, neither of the justifications $r_1 = a \leftarrow b$ nor $r_3 = a \leftarrow d$ are $posValid$ ($label(b) = unknown$, $label(d) = out$). 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 $chooseLabel$ 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 $chooseLabel(b)$ 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 $cons$, $supp$ and $label$, as well as network specific functions such as $valid$, $invalid$ and $posValid$ have been factored out in a separate class `SimpleNetwork` which derives from the trait `TruthMaintenanceNetwork`.

A more efficient variant can be found in the class `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

[1] Harald Beck, Thomas Eiter, and Christian Folie. Ticker: A System for Incremental ASP-based Stream Reasoning. *CoRR*, abs/1707.05304, 2017.

[2] Christoph Beierle and Gabriele Kern-Isberner. *Methoden wissensbasierter Systeme - Grundlagen, Algorithmen, Anwendungen (5. Aufl.).* Computational intelligence. SpringerVieweg, 2014.

[3] Gerd Brewka, Thomas Eiter, and Miroslaw Truszczyński. Answer Set Programming at a Glance. *Communications of the ACM*, 54(12):92–103, 2011.

[4] Jon Doyle. A Truth Maintenance System. *Artif. Intell.*, 12(3):231–272, 1979.

[5] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer Set Programming: A Primer. In *5th International Reasoning Web Summer School (RW 2009), Brixen/Bressanone, Italy, August 30–September 4, 2009*, pages 40–110, 2009.

[6] Charles Elkan. A Rational Reconstruction of Nonmonotonic Truth Maintenance Systems. *Artif. Intell.*, 43(2):219–234, 1990.

[7] Drew V. McDermott. A General Framework for Reason Maintenance. *Artif. Intell.*, 50(3):289–329, 1991.