ANZIAM J. 51(2009), 102–122
doi:10.1017/S1446181109000352

A SIMPLE ALGORITHM FOR DEDUCTION

BILL WHITEN[1]

(Received 8 March, 2009; revised 9 September, 2009)

Abstract

It is shown that a simple deduction engine can be developed for a propositional logic that follows the normal rules of classical logic in symbolic form, but the description of what is known about a proposition uses two numeric state variables that conveniently describe unknown and inconsistent, as well as true and false. Partly true and partly false can be included in deductions. The multi-valued logic is easily understood as the state variables relate directly to true and false. The deduction engine provides a convenient standard method for handling multiple or complicated logical relations. It is particularly convenient when the deduction can start with different propositions being given initial values of true or false. It extends Horn clause based deduction for propositional logic to arbitrary clauses. The logic system used has potential applications in many areas. A comparison with propositional logic makes the paper self-contained.

2000 *Mathematics subject classification*: primary 03B05; secondary 03B50, 03B53, 03B70.

Keywords and phrases: deduction, classical propositional logic, two-variable logic, simple algorithm.

1. Introduction

By an appropriate choice of rule format and proposition description, a simple and easily implemented algorithm for deduction can be derived. While the proposition states are described by two numeric variables, a simple interpretation in terms of true and false is directly available. It can be considered to be an extension of deduction using Horn clauses that is not restricted in the form of the clauses used. The algorithm has a wide range of potential uses. Due to its symmetrical use of the rule base it is particularly useful where information on different propositions can be supplied as the initial data. Useful student exercises can be based around implementing and evaluating versions of this algorithm. One published application using the logic and deduction algorithm of this paper is to the optimization of ultrasonic transducers [9].

[1]The University of Queensland, SMI, JKMRC, Brisbane 4072, Australia; e-mail: W.Whiten@uq.edu.au.

An application of particular interest is for decisions within computer programs. By using a rule base that takes the input conditions and evaluates indicator variables that then control the sequence of events within the program, the logical decisions can be separated from the remainder of the program. As complicated logic is a regular source of problems in computer programs, the ability to separate the logic for independent testing is a significant advantage. For this application the algorithm in this paper is significantly more capable than a decision table, although it provides a way to implement and extend decision tables, and is both less complex and more robust than the Rete algorithm [5], in that it handles inconsistent states. In some cases there may be advantage in dividing the deduction into separate stages where some results of the first stage form the input to a second stage.

There have been many methods suggested for extending Boolean logic to include more than the states true and false. For instance, [16] contains well over a thousand references. More recent reviews are given in [7, 13]. Several authors, including [1, 6, 12], have recommended a four-value logic consisting of unknown, true, false, and inconsistent as having significant benefits for deduction, as the four cases cover the possible outcomes. However, in spite of the advantages, four-valued logic is not widely known or used outside of the literature on logic. For example, [3] does not mention the use of unknown or inconsistent states, and the recent review by [8] mentions unknown only to say (in Section 2.4) that it causes problems, and suggests two ways of avoiding the problems. Ben-Ari [2, Section 1.5] mentions the possibility of an undetermined state, but then uses only true and false in his description of propositional logic. The book by Russell and Norvig [10, Chapter 7] includes an unknown but not an inconsistent state, but, as is commonly the case, inconsistent is used implicitly for formulae in proof by contradiction.

The logic system presented in this paper is formally a paraconsistent logic (see [11, 14] and references therein) in that inconsistencies are tolerated, can be identified, and are not indefinitely propagated. Paraconsistent logics are controversial and as such have developed a significant literature. Troublesome expressions such as $A \mid \neg A \mid X$ should not be generated gratuitously, and should if present be reduced to true before deduction starts.

In this paper it is shown how the normal rules of propositional logic can be retained, while a multi-valued logic is used to provide a robust deduction algorithm. The multi-valued logic is based on separate *%true* and *%false* values associated with each proposition; these can conveniently describe what is known about a proposition, and allow a very simple deduction engine. The 0% and 100% points of the *%true* and *%false* scales define a four-valued logic, and intermediate percentages can be used to indicate uncertainty in the true or false values. Although this is a multi-valued logic, being constructed using *%true* and *%false* values makes it easily understood by untrained users.

The rules can be described using the familiar **If–Then** structure, or other logical relations if required, but are transformed to a symmetric form, known as the conjunctive normal form, for the deduction engine. All the transformations of the

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

rules are made according to the usual rules of classical logic. Deduction relies on **If–Then** rules and a simple transformation to *%true* and *%false* state variables.

The deduction algorithm runs quickly with typically only a small number of evaluations of each rule required. The number of passes over the rule base depends on length of possible chains from one rule to the next and the order of the rules. Typically the length of possible chains can be expected to be low, usually below 10, and the ordering of the rules can be expected not to be such that a complete pass over the rules is required for each link in a chain of rules. A simple extension of the basic algorithm can, at the expense of some preprocessing, avoid evaluating rules in which no variables have changed (see Section 8.9). The simpler algorithm can be used until such time as it is found that a faster algorithm is required.

The following sections first describe how propositions are given values for use in deduction. This is followed by some notation, and a description of the rules and their transformation. Next the deduction engine is described, followed by an example and some implementation details. Comments and conclusions complete the paper.

2. Propositions

Deduction is about some propositions that can be true or false. In this paper propositions are always assumed to be simple, rather than compound expressions or rules which are formed by combining propositions. For example, we might have "*the algorithm needed is linear programming*" as a proposition that can be true or false. The propositions are used in rules, which give relations between the propositions and are assumed to be always true. These rules, given some initial proposition values, can be used to deduce the status of other propositions.

Classical logic assumes that a proposition is either true or false. However, a deduction engine needs also to be able to express the fact that the status of a proposition is unknown. In fact, most propositions will start from the unknown status. Also it is quite possible that by accident (or otherwise) rules and/or data can give inconsistent information about a proposition, and hence it is also desirable to be able to detect and report this situation. Thus, for describing what is known about a proposition, the logic needed is a four-valued logic, with states unknown, true, false, and inconsistent.

Further the information we have about a proposition may be uncertain. This can be conveniently expressed as a percentage. Note that if sufficient information is available to calculate probabilities, the rules of statistics should be applied. However, in many cases there is not sufficient information for a proper or even approximate statistical analysis.

A convenient way of expressing the status of knowledge about a proposition is to use two variables, one giving the extent to which it is known that the proposition is true, and the second giving the extent to which it is known that the proposition is false. It is generally convenient to express these state variables as *%true* and *%false*, although in many cases only the 0% and 100% values need to be used. Another range for these state variables can be chosen if desired.

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

In the case where the percentages are restricted to either 0% or 100% a four-valued logic results.

State	*%true*	*%false*
True	100	0
False	0	100
Unknown	0	0
Inconsistent	100	100

Comparing this with classical true/false logic, we see that true and false are easily obtained from the two state variables. Unknown as 0 *%true* 0 *%false* is easily understood, and inconsistent is also clear. It should be noted that that the numeric state variables are always defined regardless of the state of the corresponding proposition. The application of operators to these state variables is described in the following sections and some examples are given in Appendix A.

Partly true and/or partly false values can be introduced as percentages between 0 and 100 as needed. For example, a proposition given as 60 *%true* and 60 *%false* indicates there is about equal evidence for the proposition being true, and for the proposition being false, while 30 *%true* and 30 *%false* also indicates equal evidence but at a lower level, and 70 *%true* and 20 *%false* indicates more evidence for true than for false. Additional information is needed if these situations are to be further resolved. Note that all three cases above are different from no information supporting true or false which is 0 *%true* and 0 *%false*.

While this use of the two state variables to describe what is known about a simple proposition is different from classical logic, after a little practice it is generally found to be convenient and easy to use.

3. Notation

The notation used in this paper has been chosen to correspond to that used in computer programs rather than that in formal logic, as this notation is more likely to be familiar to those who might use the algorithm given in this paper. It is of course a matter of a simple change of symbols to use the notation of formal logic.

Operators such as **If** and **Then** will be given in bold type, with an initial capital letter. A | (bar) will be used for **Or** (for example, $A \mid B$ is true, if either A is true, or B is true; or both are true), and & (ampersand) will be used for **And** (for example, A & B is true, if both A and B are true). The symbol ¬ (negate) will be used for **Not**. The manner in which **Not** is interpreted is explained in due course.

We will use one or more capital letters (but not **T**, **F** or U to Z) as a short symbol that denotes a simple proposition, and the corresponding lower-case letters with superscript T or F to denote the two percentage variables (*%true* and *%false*) that define the known status of the proposition. For example, the proposition "*the algorithm needed is linear programming*" might be represented by LINPROG and its two variables defining its known status will be the values of $linprog^T$ and $linprog^F$.

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

When describing the general case where the state variable could be either a *%true* or a *%false* the superscript is omitted.

It should be carefully noted that the lower-case letters denote numeric values associated with the logical proposition but are not the logical proposition. The rules for operating on the lower-case percentage variables differ from those for the upper-case logical variables.

T and **F** are reserved to describe a proposition or logical formula as being true or false. The letters X to Z are used to denote expressions made up of propositions, and the lower-case letters x to z denote expressions made up of the state variables. The letters U to W are used to represent a simple proposition or the **Not** of a simple proposition, which are the cases that convert to a simple state variable.

For formulas that update a variable's value, the new value of the variable will be given by the expression on the left-hand side of the replacement operator := (for instance, $x := x + 1$).

The lower-case letters i, j, k are used for subscripts and m, n are used for the number of rules and the number of propositions in a rule.

While propositions are described by two state values (*%true* and *%false*), expressions involving the state variables have only a single numeric value that is determined by the variables in the expression. The method used to evaluate expressions is described in the section on deduction.

4. Rules

Rules provide the known relations between the propositions or, in the case of an expert system, the expert knowledge. They are used to make deductions about the propositions typically using initial values for some of the propositions. It is assumed that each rule is true. Section 8.5 indicates how partly true rules can be accommodated.

Rules are generally easiest developed and understood in the form

$$\textbf{If } U \ \& \ \cdot \ \cdot \ \cdot \ \& \ V \textbf{ Then } W, \tag{4.1}$$

where the propositions (U to W) can be a simple proposition name indicating true is to be used when applying the rule, or can be in the form $\neg A$, where A is a simple proposition name, indicating false is used (for instance, a rule could be **If** $\neg A \ \& \ B$ **Then** $\neg C$). Several rules of this form are assumed to have an implicit **And** between them. Thus all the given rules are assumed to be true, and are available for use in deduction.

Any logical conditions can be converted to multiple **If–And–Then** rules of the type above. The well-known rules of propositional logic can be used to do this (Appendix B gives these rules). For instance,

$$\textbf{If } A \mid B \textbf{ Then } C$$

is equivalent to the two rules

$$\textbf{If } A \textbf{ Then } C \quad \text{and} \quad \textbf{If } B \textbf{ Then } C.$$

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

It should be noted that the rules in terms of the symbolic propositions follow the normal rules of propositional logic, while the deductions operate on the state variables describing the current knowledge about the propositions.

A rule of the form

$$\textbf{If } X \textbf{ Then } Y \tag{4.2}$$

can be written as

$$\neg X \mid Y \quad \text{is true} \tag{4.3}$$

which says that either Y is true, or $\neg X$ is true, or both are true (see Appendix B for more details). Note that in this form the rule is symmetric in $\neg X$ and Y.

For the rule **If** X **Then** Y (that is, $\neg X \mid Y$ is true), if $\neg Y$ is true then $\neg X$ must be true, for if X is true we get Y is true which is inconsistent. This can be written as

$$\textbf{If } \neg Y \textbf{ Then } \neg X$$

which can also be written as $\neg X \mid Y$ is true.

Similarly the condition

$$A_1 \mid A_2 \mid \cdots \mid A_n \quad \text{is true} \tag{4.4}$$

corresponds to the n rules

$$\textbf{If } Z \textbf{ Then } A_k, \tag{4.5}$$

where Z is the **And** of the $\neg A_j$ excluding $\neg A_k$ (which is also the **Not** of the **Or** of the A_j excluding A_k, and thus this is derived from the equality of the rule (4.2) and expression (4.3)).

So we see that one **If–And–Then** rule (as in (4.1)) converts to a symmetric **Or** format that, if there are n terms, supplies n different **If–And–Then** rules that are actually equivalent in terms of propositional logic. While the **Or** format for rules is convenient for deduction, in that it is symmetric and converts easily to each of the equivalent **If–Then** expressions, most people seem to prefer thinking and working with the **If–Then** format. Fortunately the transformation to the **Or** format is purely mechanical, and it is both possible and convenient to prepare rules in the **If–And–Then** format. Converting all the rules to the form (4.4) gives the conjunctive normal form.

Finally, we note several rules with the same variable after the **Then** (that is, in the **Or** format (4.4) the rules contain the same variable) can be combined into one rule; for example, the m rules

$$\textbf{If } X_1 \textbf{ Then } W, \quad \textbf{If } X_2 \textbf{ Then } W, \quad \ldots, \quad \textbf{If } X_m \textbf{ Then } W$$

can be written as

$$\textbf{If } X_1 \mid X_2 \mid \cdots \mid X_m \textbf{ Then } W. \tag{4.6}$$

This then provides a single rule for deduction of W from the available rules about W. As indicated above, it will be assumed that the rules have been transformed so that each X_i is of the form $U_1 \,\&\, U_2 \,\&\, \cdots \,\&\, U_n$ where U_j is either a simple proposition or the **Not** of a simple proposition, and W is also of the form A or $\neg A$.

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

5. Deduction

Deduction is done by finding rules that, given the current values of some of the propositions, can only be made true by giving a particular value to another proposition. This occurs in an obvious manner for the proposition following the **Then** in a rule having the **If**–**Then** format. Deduction in this paper will be done by evaluating, using the state variables, the expression between **If** and **Then**, and giving its value to the state variable of the proposition (or **Not** proposition) following the **Then**. This provides a clear interpretation for deduction that extends to using partly true propositions.

However, as seen above, one rule gives rise to several different **If**–**Then** rules. For symmetry and convenience (of the algorithm rather than the user) the **Or** form of the rules is used within the deduction engine.

For deduction the proposition state variables are substituted into the rules. This is done by replacing the **Not** of a proposition ($\neg A$) by the corresponding *%false* value a^F, and replacing a simple proposition A by its *%true* value a^T. Note that for consistency this requires that $\neg a^T$ can be replaced by a^F, and $\neg a^F$ can be replaced by a^T. This substitution expresses the rules in terms of the known states of the propositions. Note that there are no longer any **Not** operators in these rules, and expressions can always be evaluated as the state variables are always defined.

Having made this substitution, the state variables can be treated as independent numeric values. It is easily seen that transforming an expression and then converting to the state variables usually gives the same result as converting to the state variables and then making the transformation. For instance, for the transformation step applying **Not** to the expressions $A \mid B \mid C$ and $a^T \mid b^T \mid c^T$ gives $\neg A \,\&\, \neg B \,\&\, \neg C$ and $a^F \,\&\, b^F \,\&\, c^F$. Appendix C summarizes the difference between symbolic manipulation of propositions (as given in Appendix B) and evaluation of expressions containing the state variables.

The deduction engine needs to evaluate the **And** and **Or** operators that occur between the **If** and **Then** terms in expressions similar to those in rules (4.5) and (4.6). For these we choose the minimum and maximum:

$$a \,\&\, b \quad \text{is evaluated as} \quad \min(a, b),$$
$$a \mid b \quad \text{is evaluated as} \quad \max(a, b).$$

Thus both the state variables and expressions involving them evaluate to numeric values. These are the same rules that are often used in fuzzy set theory and various other applications. They agree with normal logic for propositions that are true or false, and are easily seen to give the expected results when applied to unknown values (Appendix A gives some examples). They have convenient properties for calculation, in that they are simple to apply, and if applied multiple times the result is not changed. This last property means that rules can be applied in any order and repeated, without the deductions giving a different result.

Values of the *%true* or *%false* variables between 0 and 100 indicate uncertainty about the proposition. The min/max rules give an approximate way of dealing with uncertainty which may be adequate for some applications.

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

The deduction engine needs to handle both the multiple **If–Then** deductions possible from one rule in **Or** format, and also handle the results of several rules that contain the same variable. The following subsections describe deduction from a single rule, the deduction from multiple rules, and finally the complete deduction engine.

5.1. Single rules For the deductions possible from a single rule we note that there are only two values that a variable can receive from the rule. These depend on which variable is placed on the **Then** side of the rule. The rule

$$a_1 \mid a_2 \mid \cdot \cdot \cdot \mid a_n \tag{5.1}$$

written, for instance, as

$$\textbf{If} \neg a_2 \ \& \ \cdot \ \cdot \ \cdot \ \& \neg a_n \ \textbf{Then} \ a_1$$

sets a_1 to the value $\min(\neg a_j \ ; \ \ j = 2 : n)$ (remember for the proposition A_j that a_j is one of the state variables and $\neg a_j$ is the other state variable). This is the same as $\min(\neg a_j \ ; \ \ j = 1 : n)$ if $\neg a_1$ is not the minimum, and if $\neg a_1$ is the minimum of $(\neg a_j \ ; \ \ j = 1 : n)$ then the value for a_1 is the second smallest of $(\neg a_j \ ; \ \ j = 1 : n)$.

Hence for the rule (5.1) the value of a_k is set to $\min(\neg a_j \ ; \ \ j = 1 : n)$ unless $\neg a_k$ is the minimum value, in which case a_k is set to the second smallest of the $(\neg a_j \ ; \ \ j = 1 : n)$. Note that the second smallest value may be equal to the smallest value. That is, the deduction calculation for a single rule (5.1) is

$$\begin{aligned}
&\alpha := \min(\neg a_j \ ; \ \ j = 1 : n) \\
&\beta := \text{secondsmallest}(\neg a_j \ ; \ \ j = 1 : n) \\
&\textbf{for } j := 1 : n \\
&\quad a_j := \textbf{if } \neg a_j = \alpha \textbf{ then } \beta \textbf{ else } \alpha \\
&\textbf{end } \textit{for } j.
\end{aligned}$$

Minor variations of this can also be used: for instance, the test condition could use the subscript of the smallest value.

5.2. Multiple rules Multiple rules containing the same variable can be written as (from (4.6))

$$\textbf{If } b_1 \mid b_2 \mid \cdot \cdot \cdot \mid b_m \textbf{ Then } c,$$

where b_i is the value determined (as in the previous subsection) for c from the ith rule containing c. The value of c is $\max(b_i \ ; \ \ i = 1 : m)$, and this can be calculated incrementally as

$$\begin{aligned}
&c := 0; \\
&\textbf{for } i = 1 : m \\
&\quad c := \max(c, b_i) \\
&\textbf{end } \textit{for } i.
\end{aligned}$$

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

It does not matter in what order the rules are applied or whether a rule is applied more than once, as the maximization over all the rules ensures that the resulting value of c will be the same.

5.3. Deduction engine We can now combine the deduction for a single rule with that for multiple rules. The rules are assumed to have been converted to the **Or** format giving the m rules

$$a_{i,1} \mid a_{i,2} \mid \cdot \cdot \cdot \mid a_{i,n_i} \quad ; \; i = 1 : m. \tag{5.2}$$

Initial values are provided for the state variables for each proposition, some variables will be given values corresponding to the initially known information, and the remainder of the state variables are set to zero. The deduction algorithm combines the deduction for a single rule and that for multiple rules to give:

Repeat *until no further changes occur*
  for $i = 1 : m$
    $\alpha := \min(\neg\, a_{i,j} \; ; \;\; j = 1 : n_i)$
    $\beta := \text{secondsmallest}(\neg\, a_{i,j} \; ; \;\; j = 1 : n_i)$
    for $j = 1 : n_i$
      $a_{i,j} := \max(a_{i,j}, \;$ **if** $\neg\, a_{i,j} = \alpha$ **then** β **else** $\alpha)$
    end *for j*
  end *for i*
end *repeat*.

This algorithm repeatedly uses each rule to deduce the values of the variables in the rule, and if the value is larger than the current value of the variable, saves the new value. The algorithm will always converge as the $a_{i,j}$ can only take values that are given initially to some of the $a_{i,j}$, and $a_{i,j}$ can only increase.

6. Example

To see how the deduction algorithm operates it is suggested that the reader work through some simple examples. It is often helpful to mentally or otherwise convert the rules back to the various implied **If**–**And**–**Then** rules to assist in understanding the algorithm steps. The following five rules:

1	**If** QUACKS **And** WADDLES	**Then** DUCK
2	**If** JUMPS **And** CROAKS	**Then** FROG
3	**If** DUCK	**Then Not** FROG
4	**If Not** CROAKS	**Then** QUACKS
5	**If Not** WADDLES **And Not** JUMPS	**Then** WALKS

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

convert to the following conditions using the *%true* and *%false* state variables.

Rule 1 $quacks^F \mid waddles^F \mid duck^T$ Rule 2 $jumps^F \mid croaks^F \mid frog^T$

Rule 3 $duck^F \mid frog^F$ Rule 4 $croaks^T \mid quacks^T$

Rule 5 $waddles^T \mid jumps^T \mid walks^T$

Starting with the information: does not walk, waddle, or quack, which, using the state variables, becomes

$$walk^F\ 100 \quad waddles^F\ 100 \quad quacks^F\ 100,$$

deduction then proceeds with the following steps (in each pass all of the rules are evaluated in turn, however for brevity after the first pass only rules that result in a change have been recorded below):

Pass 1, Rule 1	no change	Pass 1, Rule 2	no change
Pass 1, Rule 3	no change	Pass 1, Rule 4	$croaks^T\ 100$
Pass 1, Rule 5	$jumps^T\ 100$		
Pass 2, Rule 2	$frog^T\ 100$	Pass 2, Rule 3	$duck^F\ 100$
Pass 3	no changes		

Hence the result of this deduction is FROG true, and DUCK false.

A second example is given in Appendix D. Again the reader in invited to work through the example step by step to see how the algorithm operates.

7. Implementation

The implementation of the algorithm requires the storage of the rules and the storage of the two state variables for each proposition. For the percentage scale one byte is all that is needed for each variable. The ease and efficiency of implementation depend largely on the method of storage used and on how $\neg x$ is determined from x.

One convenient scheme is to store the state variables in an array with the *%true* values in the even locations and the corresponding *%false* in the following location. Conversion from x to $\neg x$ as required in the algorithm can then be done with a bitwise exclusive or of the location in the array with one, which turns an even subscript to the following odd subscript, and an odd subscript to the preceding even subscript.

The rules can then be stored in a sparse matrix form, as a list of vectors that give the subscripts of the variables in the **Or** form of each rule.

Other storage schemes can be envisaged and a useful exercise is to compare the different possible schemes.

8. Comments

This deduction engine, although simple to implement, has a range of interesting properties that are discussed in the following subsections.

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

8.1. Preparation for deduction The deduction proceeds using three stages, of which the first two prepare the rules for input to the deduction engine and the last is the actual deduction. The first stage converts the rules to a standard form, which uses only **Or** and **Not** applied to simple propositions, by applying classical propositional logic (Appendix B). The next stage converts the rules to contain only the proposition state variables (*%true* and *%false*). This stage removes all the **Not** operators from the rules. The last stage is the deduction engine where the numeric values of the state variables are combined with the rules to make deductions.

8.2. Propositions, variables and rules The normal rules of logic are applied to propositions, while what is known about a proposition is given by the two state variables. For deduction, rules and formulae are expressed using state variables which are simple scalar values. This creates a difference between the symbolic expressions of Boolean algebra that assume a proposition is either true or false, and the numeric evaluation that uses what is actually known about the proposition. Numeric evaluation of a formula, including the trivial formulae a^T and a^F, always gives a single scalar value, and only one of a proposition's state variables is determined by a single rule. This provides a convenient mechanism to allow propositions to be unknown, party known, or inconsistent (in the case of inconsistent information).

A formula X can be symbolically equated to a proposition A which creates the two rules

$$\textbf{If } X \textbf{ Then } A, \quad \textbf{If } \neg X \textbf{ Then } \neg A. \tag{8.1}$$

In this manner an expression can be converted back to a proposition with the two state variables. Without this equivalence with a simple proposition which needs the two parts, expressions evaluate from the state variables to a simple numeric value, which does not have an associated *%false* value unless the **Not** of the expression is also evaluated.

8.3. Repeated rules The use of minimum and maximum for **And** and **Or** allows the deduction algorithm to proceed without checking whether a rule has been used multiple times. This simplifies the deduction algorithm. It is also convenient in formulating the rules where it may not be easy to determine if a rule is redundant, directly or as a combination of other rules. Redundant rules could slightly reduce or increase the algorithm efficiency but do not have a detrimental effect on the result. The next subsection shows how certain formally redundant rules can sometimes be added to extend the deduction power of the algorithm.

8.4. Additional deductions Determining the values of logical propositions that satisfy given logical relations is a typical example of an NP-complete (nondeterministic polynomial [4, 15]) problem for which no efficient algorithms are known, and solutions can expand exponentially in the size of the problem. The deduction algorithm of Section 5.3 proceeds by passing information using the state variables of the propositions, however some deductions cannot be made in this manner. It is this limitation that allows the simple deduction algorithm. In some cases further

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

deductions can be made by passing information in a combination of propositions. The number of possible combinations can increase exponentially with the number of rules. Where necessary, additional rules can be added to allow any required extra deductions to be made.

For instance, for the rules

$$\textbf{If} \neg B \ \& \neg C \ \textbf{Then} \ A, \quad \textbf{If} \ B \ \textbf{Then} \ D, \quad \textbf{If} \ C \ \textbf{Then} \ D \tag{8.2}$$

with the data A is false, using the above deduction algorithm does not derive D as true. Here if the rules are rewritten as

$$\textbf{If} \neg A \ \textbf{Then} \ (B \mid C), \quad \textbf{If} \ (B \mid C) \ \textbf{Then} \ D, \tag{8.3}$$

we see that the deduction from $\neg A$ to D is transmitted via the expression $B \mid C$ rather than as the value of a single variable. This expression can be replaced in (8.3) by a simple expression that is defined using the relations (8.1). Alternatively and more simply, the additional rule

$$\textbf{If} \neg A \ \textbf{Then} \ D$$

can be added to the rules (8.2), giving for deduction the four rules

$$a^T \mid b^T \mid c^T, \quad b^F \mid d^T, \quad c^F \mid d^T, \quad a^T \mid d^T.$$

Both options are sufficient to allow the missing deduction to be made.

It is also possible to use a more complete method of deduction with the proposition state defined by the *%true* and *%false* variables in a straightforward manner. However the more complete deduction methods may require considerably more computation, particularly when, as in the proposed algorithm, deductions about all the propositions are requested.

8.5. Partly true rules If a partly true rule is needed it is easily implemented by introducing into the rule a partly true proposition that describes the extent to which the rule is true. For instance, the rule **If** A **Then** B is changed to **If** A & P **Then** B where P is the variable that determines how true the rule is. The *%false* value of this variable gives the extent to which the rule can be deduced as not true.

8.6. Extending Horn clauses A Horn clause is of the form [10]

$$\textbf{If} \ A \ \& \ B \ \& \ \cdots \ \& \ D \ \textbf{Then} \ E,$$

where A to E are simple propositions (or E can be the constant false, that is, **F**) and the clause contains no negation operators. This is a similar form to that used for deduction after conversion to only *%true* and *%false* variables. However, unlike forward chaining using Horn clauses, both true and false values can be deduced and all of the multiple forms of the **If–And–Then** rules are used for deduction.

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

8.7. Excluded middle The logic rule excluding values other than true and false ($X \mid \neg X = \mathbf{T}$) has long been controversial. For this deduction engine $X \mid \neg X$, being symbolic, is assumed to be true regardless of the state of knowledge about X, which could be unknown. Having converted to the state variables the corresponding $x^T \mid x^F$ no longer has the corresponding value of 100%. This is because state variables describe what is known about the proposition, rather than a logical requirement based on the proposition being either true or false. A rule $A \mid \neg A$ becomes **If** a^T **Then** a^T and also **If** a^F **Then** a^F, which while clearly true are not useful.

Similarly, the expression $X \,\&\, \neg X$ is treated as false for symbolic calculations, but when evaluated using the state variables $x^T \;\&\; x^F$ may well give values greater the zero, indicating some level of inconsistency. The propagation of inconsistencies can be reduced if deductions are restricted to giving numeric values greater than the numeric value of the **Or** form of the rule (expressions (4.4) and (5.2)).

8.8. Fuzzy set logic A fuzzy set description of the logic values can be incorporated by using a cumulative description of the fuzzy set to describe *%true* and also *%false*. Then $c := a \mid b$ is evaluated using the inverse cumulative fuzzy distribution as

$$\mathrm{cdf}_c^{-1}(t) := \max(\mathrm{cdf}_a^{-1}(t),\, \mathrm{cdf}_b^{-1}(t)),$$

where t goes from zero to one or takes certain discrete values between zero and one. Similarly, for $c := a \;\&\; b$,

$$\mathrm{cdf}_c^{-1}(t) := \min(\mathrm{cdf}_a^{-1}(t),\, \mathrm{cdf}_b^{-1}(t)).$$

When the cumulative distribution functions are a step function from zero to one these reduce to the simple *%true* and *%false* case.

8.9. Efficiency In most cases the algorithm given requires only a few iterations and the amount of computation needed is not a problem. Each iteration takes a time proportional to the total number of proposition mentions in the rules. However, the algorithm can be made more efficient in several ways, including by calculating α and β in the same loop and also noting that if α is zero the inner deduction loop can be skipped. A reordering of the rules can reduce the number of iterations needed.

For large rule bases with few rules active it may be more efficient to only evaluate the rules containing state variables that have changed in value. This requires some preprocessing of the data to create a table giving the rules that can produce new deductions for a change in each of the state variables. Then an array containing one element for each rule can contain zero if the rule does not need to be evaluated, or elements of a linked list of the rules pending evaluation. Rules are evaluated from the beginning of the linked list until the list becomes empty. When a state variable changes value, the rules that use that state variable (that is, $\neg a_{i,j}$ from expression (5.2)) are added to the end of the list if they are not already on the list.

It is also possible to maintain and update the values of α and β for each rule as variables used by the rules are updated. This adds another portion of complexity to the algorithm and, depending on the particular rule base, may or may not give a benefit.

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

9. Conclusions

For symbolic manipulation of rules the well-known formulae of propositional logic are used to transform rules (or any other logical condition) to the **Or** format, where each rule contains only the **Or** binary operator with the **Not** operator preceding some of the propositions, and there is an implied **And** between the rules (conjunctive normal form). For the most common rule format (**If–And–Then**) this transformation is trivial.

Two numeric state variables are used to describe what is currently known about a proposition. One, the *%true*, gives the extent of evidence that the proposition is true, and the other, the *%false*, gives the extent of evidence that the proposition is false. These state variables are always defined and thus expressions containing them can always be evaluated to a numeric percentage.

The deduction engine, which is described in only a few lines of code, is based on deduction using the multiple **If–Then** rules that come from rules in the **Or** format. Evaluation of expressions containing the state variables is done using minimum for **And**, and maximum for **Or**. This allows both redundant rules and multiple evaluations of the same rule, thus simplifying both rule development and the deduction algorithm.

The use of *%true* and *%false* to describe the known status of a logical proposition has been readily accepted by users of this deduction engine. This logic system has potential uses beyond the deduction algorithm in this paper.

Several extensions of the basic algorithm are possible including: partly true rules, use of fuzzy set based logic, and more efficient calculation.

A Matlab implementation of this algorithm is available from:

`http://www.mathworks.com/matlabcentral/fileexchange/9218`

or can be found by a search for Matlab EXPERT1.

Acknowledgements

The assistance of Graham Sheridan in checking parts of an earlier version of this paper is gratefully acknowledged. Useful suggestions from the referees are acknowledged.

Appendix A. Result tables for *%true* and *%false* logic

Table 1 gives how the operator **Not** is interpreted when used in an expression and in rules. This is followed by sample tables for the operators **Or** (Table 2) and **And** (Table 3). Note that in these tables both state variables are given for completeness, although only the value of one is used in the evaluation of the expression. The other tables for these operators are easily constructed.

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

TABLE 1. Result table for **Not** in expressions and rules.

Proposition			In expressions		**If T Then** X Updated X		**If T Then** $\neg X$ Updated X	
State	x^T	x^F	$X = x^T$	$\neg X = x^F$	x^T	x^F	x^T	x^F
Unknown	0	0	0	0	100	0	0	100
False	0	100	0	100	100	100	0	100
True	100	0	100	0	100	0	100	100
Inconsistent	100	100	100	100	100	100	100	100

TABLE 2. Result table for operator **Or** in $X \mid Y$ (which becomes the numeric value $x^T \mid y^T$ given in the lower right partition).

		y^T, y^F			
$X \mid Y$ $(x^T \mid y^T)$		0, 0	0, 100	100, 0	100, 100
x^T, x^F	0, 0	0	0	100	100
	0, 100	0	0	100	100
	100, 0	100	100	100	100
	100, 100	100	100	100	100

TABLE 3. Result table for operator **And** in $X \,\&\, \neg\, Y$ (which becomes the numeric value $x^T \,\&\, y^F$ given in the lower right partition).

		y^T, y^F			
$X \,\&\, \neg\, Y$ $(x^T \,\&\, y^F)$		0, 0	0, 100	100, 0	100, 100
x^T, x^F	0, 0	0	0	0	0
	0, 100	0	0	0	0
	100, 0	0	100	0	100
	100, 100	0	100	0	100

Appendix B. Rules for classical propositional logic

The basic identities of propositional logic are given below and should be compared with the rules that apply to the state variables *%true* and *%false* given in Appendix C.

For operator **Not** ($\neg$):

$$\neg\,(\neg\, X) \equiv X,$$
$$\neg\,\mathbf{T} \equiv \mathbf{F}, \quad \neg\,\mathbf{F} \equiv \mathbf{T}.$$

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

For operator **And** (&):

$$X \,\&\, X \equiv X, \quad X \,\&\, Y \equiv Y \,\&\, X, \quad X \,\&\, \mathbf{T} \equiv X, \quad X \,\&\, \mathbf{F} \equiv \mathbf{F},$$
$$X \,\&\, (Y \,\&\, Z) \equiv (X \,\&\, Y) \,\&\, Z \equiv X \,\&\, Y \,\&\, Z,$$
$$\mathbf{F} \,\&\, \mathbf{F} \equiv \mathbf{F}, \quad \mathbf{F} \,\&\, \mathbf{T} \equiv \mathbf{F}, \quad \mathbf{T} \,\&\, \mathbf{F} \equiv \mathbf{F}, \quad \mathbf{T} \,\&\, \mathbf{T} \equiv \mathbf{T}.$$

For operator **Or** (|):

$$X \mid X \equiv X, \quad X \mid Y \equiv Y \mid X, \quad X \mid \mathbf{F} \equiv X, \quad X \mid \mathbf{T} \equiv \mathbf{T},$$
$$X \mid (Y \mid Z) \equiv (X \mid Y) \mid Z \equiv X \mid Y \mid Z,$$
$$\mathbf{F} \mid \mathbf{F} \equiv \mathbf{F}, \quad \mathbf{F} \mid \mathbf{T} \equiv \mathbf{T}, \quad \mathbf{T} \mid \mathbf{F} \equiv \mathbf{T}, \quad \mathbf{T} \mid \mathbf{T} \equiv \mathbf{T}.$$

For combinations of & and |:

$$X \,\&\, (Y \mid Z) \equiv (X \,\&\, Y) \mid (X \,\&\, Z),$$
$$X \mid (Y \,\&\, Z) \equiv (X \mid Y) \,\&\, (X \mid Z).$$

And for combinations including $\neg$:

$$X \,\&\, \neg X \equiv \mathbf{F}, \quad X \mid \neg X \equiv \mathbf{T},$$
$$\neg (X \,\&\, Y) \equiv \neg X \mid \neg Y, \quad \neg (X \mid Y) \equiv \neg X \,\&\, \neg Y.$$

Proofs can be performed by application of the basic formulae above. Alternatively, as the propositions can be restricted to true or false values, an enumeration of all possible values can be done. If two expressions have the same values for all possible inputs the expressions are identical, for instance:

X	Y	$\neg X \mid Y$	**If** X **Then** Y
F	**F**	**T**	**T**
F	**T**	**T**	**T**
T	**F**	**F**	**F**
T	**T**	**T**	**T**

Thus:

$$(\mathbf{If}\ X\ \mathbf{Then}\ Y) \equiv (\neg X \mid Y).$$

Statements of the form **If** X **Then** Y when X is false often require an explanation. Firstly, in making deductions it is required that all the given rules are made true. Looking at the possible cases where the rule is assumed to be false gives:

> if the rule **If** X **Then** Y is false when X is false (regardless of the value of Y), this would require X to be true to make the rule true—clearly not the meaning required for the rule;

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

if the rule **If** X **Then** Y is false when X and Y are false, and true when X is false and Y is true, X false would require Y true to make the rule true—again not the meaning required;
similarly, if the rule **If** X **Then** Y is true when X and Y are false, and false when X is false and Y is true, X false would require Y to be false.

Hence the only way this rule does not give deductions contrary to its conventional meaning is to have **If** X **Then** Y true when X is false as in the above table. When X is false no deduction is made as the rule is already true.

Further identities involving **If**–**Then** can be derived:

$$
\begin{gathered}
(\ \textbf{If}\ X\ \textbf{Then}\ Y) \equiv (\ \textbf{If}\ \neg\, Y\ \textbf{Then}\ \neg\, X\ ),\\
\textbf{If}\ (\ \textbf{If}\ X\ \textbf{Then}\ Y)\ \&\ (\ \textbf{If}\ Y\ \textbf{Then}\ Z)\ \textbf{Then}\ (\ \textbf{If}\ X\ \textbf{Then}\ Z),\\
(\ \textbf{If}\ X\ \textbf{Then}\ Z\ )\ \&\ (\ \textbf{If}\ Y\ \textbf{Then}\ Z\ ) \equiv (\ \textbf{If}\ X\ \mid\ Y\ \textbf{Then}\ Z\ ),\\
(\ X \equiv Y\ ) \equiv (\ \textbf{If}\ X\ \textbf{Then}\ Y\ )\ \&\ (\ \textbf{If}\ \neg\, X\ \textbf{Then}\ \neg\, Y\ ).
\end{gathered}
$$

Appendix C. Rules for deduction

Deduction in this paper is done using rules in the form **If** X **Then** U with the value X being evaluated using

$$a\ \&\ b \equiv \min(a, b), \quad a \mid b \equiv \max(a, b) \tag{C.1}$$

after expressing X in terms of the *%true* and *%false* state variables. This value then sets a state variable for U which is required to be a simple proposition, or the **Not** of a simple proposition. It is again noted that the lower-case state variables represent numeric values rather than logical propositions, and both these variables and expressions evaluate to a purely numeric value. The use of percentages has been chosen to emphasize this. It is probably more straightforward to convert the logical expressions to the required form before introducing the numeric state variables for evaluation. However, the following gives rules for symbolic manipulation and numeric evaluation of the state variables.

Expressions of state variables involving **Not** can be handled by converting back to an expression using the propositions, transforming using the above identities so that the **Not** operates on simple propositions, and then converting back to the state variables. For instance:

$$\neg\,(a^T \mid b^F \mid c^T)\ \rightarrow\ \neg\,(A \mid \neg\, B \mid C)\ =\ \neg\, A\ \&\ B\ \&\ \neg\, C\ \rightarrow\ a^F\ \&\ b^T\ \&\ c^F.$$

It is easy to verify that the interpretation of **And** and **Or** as minimum and maximum, together with $\mathbf{T} = 100\%$ and $\mathbf{F} = 0\%$, satisfies the rules for **And** and **Or** given in Appendix B. A direct evaluation of **Not** is not defined; however, **Not** can be defined as an operator that converts to expressions that can be evaluated. Rules for the direct transformation of expressions using the state variables can be derived. These,

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

not surprisingly, are similar to those for propositional logic, with the exception that the four identities

$$\neg \mathbf{T} \equiv \mathbf{F}, \quad \neg \mathbf{F} \equiv \mathbf{T}, \quad X \,\&\, \neg X \equiv \mathbf{F} \quad \text{and} \quad X \mid \neg X \equiv \mathbf{T}$$

do not convert to state variables, as these involve corresponding *%true* and *%false* state variables which are independent. The conversion rules $\neg a^T = a^F$ and $\neg a^F = a^T$ replace these four rules.

Thus the rules that can be used for manipulation of expressions of the numeric state variables are:

$$\neg a^T \equiv a^F, \quad \neg a^F \equiv a^T, \tag{C.2}$$

$$x \,\&\, x \equiv x, \quad x \,\&\, y \equiv y \,\&\, x, \quad x \,\&\, 100\% \equiv x, \quad x \,\&\, 0\% \equiv 0\%,$$
$$x \,\&\, (y \,\&\, z) \equiv (x \,\&\, y) \,\&\, z \equiv x \,\&\, y \,\&\, z,$$
$$x \mid x \equiv x, \quad x \mid y \equiv y \mid x, \quad x \mid 0\% \equiv x, \quad x \mid 100\% \equiv 100\%,$$
$$x \mid (y \mid z) \equiv (x \mid y) \mid z \equiv x \mid y \mid z,$$
$$x \,\&\, (y \mid z) \equiv (x \,\&\, y) \mid (x \,\&\, z), \quad x \mid (y \,\&\, z) \equiv (x \mid y) \,\&\, (x \mid z),$$
$$\neg (x \,\&\, y) \;\equiv\; \neg x \mid \neg y, \quad \neg (x \mid y) \;\equiv\; \neg x \,\&\, \neg y, \tag{C.3}$$
$$(\textbf{If } x \textbf{ Then } y) \;\equiv\; \neg x \mid y.$$

Similarly to the rest of this paper x, y and z can be expressions, while a^T and a^F are simple numeric variables. For evaluation, the two rules (C.1) are used. As there is no rule for evaluation of **Not** to a numeric value the above rules (particularly (C.2) and (C.3)) need to be applied to remove all **Not** operators before a numeric value can be obtained.

Appendix D. Example

As a simple example consider a few of the options for optimization programs. First the type of target to be minimized.

LINTARGET	The target is a linear expression
QUADTARGET	The target is a quadratic function
SUMSQUARES	The target is a sum of squares
OTHERTARGET	The target is not one of the above

These four propositions are mutually exclusive, giving the following rules:

1	**If** LINTARGET	**Then** ¬ QUADTARGET
2	**If** LINTARGET	**Then** ¬ SUMSQUARES
3	**If** LINTARGET	**Then** ¬ OTHERTARGET
4	**If** QUADTARGET	**Then** ¬ SUMSQUARES
5	**If** QUADTARGET	**Then** ¬ OTHERTARGET
6	**If** SUMSQUARES	**Then** ¬ OTHERTARGET

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

Next we present some propositions giving the type of constraints.

NOCONSTR	No constraints
LININEQ	Linear inequality constraints
OTHERCONSTR	Other constraints

Again these are mutually exclusive, giving the following rules:

7	**If** NOCONSTR	**Then** ¬ LININEQ
8	**If** NOCONSTR	**Then** ¬ OTHERCONSTR
9	**If** LININEQ	**Then** ¬ OTHERCONSTR

Finally, to maintain this example at a reasonable size, we give only a few of the possible types of optimization programs:

LINPROG	The problem is a linear programming problem
QUADPROG	The problem is a quadratic programming problem
LEASTSQ	The problem is one of least squares without constraints

These are related by the following rules to the propositions defined above:

10	**If** LINTARGET & LININEQ	**Then** LINPROG
11	**If** QUADTARGET & LININEQ	**Then** QUADPROG
12	**If** SUMSQUARES & NOCONSTR	**Then** LEASTSQ

As these are in fact equivalence relations the following additional rules apply:

13	**If** LINPROG	**Then** LINTARGET
14	**If** LINPROG	**Then** LININEQ
15	**If** QUADPROG	**Then** QUADTARGET
16	**If** QUADPROG	**Then** LININEQ
17	**If** LEASTSQ	**Then** SUMSQUARES
18	**If** LEASTSQ	**Then** NOCONSTR

Rules 10–12 are in the form of a decision table; however, rules that the column entries are mutually exclusive and that the rows are an equivalence have been added. This type of conversion from a decision table could well be automated and the algorithm in this paper is not limited to decision table type problems.

The rules above convert to the following using the *%true* and *%false* state variables.

Rule 1	$lintarget^F \mid quadtarget^F$	Rule 2	$lintarget^F \mid sumsquares^F$
Rule 3	$lintarget^F \mid othertarget^F$	Rule 4	$quadtarget^F \mid sumsquares^F$
Rule 5	$quadtarget^F \mid othertarget^F$	Rule 6	$sumsquares^F \mid othertarget^F$
Rule 7	$noconstr^F \mid linineq^F$	Rule 8	$noconstr^F \mid otherconstr^F$
Rule 9	$linineq^F \mid otherconstr^F$	Rule 10	$lintarget^F \mid linineq^F \mid linprog^T$
Rule 11	$quadtarget^F \mid linineq^F \mid quadprog^T$		
Rule 12	$sumsquares^F \mid noconstr^F \mid leastsq^T$		
Rule 13	$linprog^F \mid lintarget^T$	Rule 14	$linprog^F \mid linineq^T$
Rule 15	$quadprog^F \mid quadtarget^T$	Rule 16	$quadprog^F \mid linineq^T$
Rule 17	$leastsq^F \mid sumsquares^T$	Rule 18	$leastsq^F \mid noconstr^T$

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

Starting with the data

$$linprog^T \quad 100,$$

deduction proceeds with the following steps:

Pass 1, Rule 13	$lintarget^T$	100	Pass 1, Rule 14	$linineq^T$	100
Pass 2, Rule 1	$quadtarget^F$	100	Pass 2, Rule 2	$sumsquares^F$	100
Pass 2, Rule 3	$othertarget^F$	100	Pass 2, Rule 7	$noconstr^F$	100
Pass 2, Rule 9	$otherconstr^F$	100	Pass 2, Rule 15	$quadprog^F$	100
Pass 2, Rule 17	$leastsq^F$	100			
Pass 3	no changes				

Thus it is determined that a linear target and linear inequalities are needed, and the remaining propositions are all false.

In a second example to demonstrate the effect of inconsistent data, the initial data suggests that the result might be quadratic programming by setting 50% for the value for $quadprog^T$ but provides other data inconsistent with this. The data for this example is:

$$quadprog^T \ 50 \quad linineq^T \ 100 \quad lintarget^T \ 100.$$

For this input data case the deduction steps are:

Pass 1, Rule 1	$quadtarget^F$	100	Pass 1, Rule 2	$sumsquares^F$	100
Pass 1, Rule 3	$othertarget^F$	100	Pass 1, Rule 7	$noconstr^F$	100
Pass 1, Rule 9	$otherconstr^F$	100	Pass 1, Rule 10	$linprog^T$	100
Pass 1, Rule 15	$quadtarget^T$	50	Pass 1, Rule 15	$quadprog^F$	100
Pass 1, Rule 17	$leastsq^F$	100			
Pass 2	no changes				

This case is identified as a linear programming problem, but additionally two propositions are identified as partly inconsistent, namely QUADPROG with 50 *%true* & 100 *%false*, and QUADTARGET also with 50 *%true* & 100 *%false*.

References

[1] N. D. Belnap, "A useful four-valued logic", in: *Modern uses of multiple-valued logic*, (eds J. M. Dunn and G. Epstein), (D Reidel Publishing, Dordrecht, Holland, 1975) 8–37.

[2] M. Ben-Ari, *Mathematical logic for computer science* (Springer, London, 2004).

[3] D. M. Bourg and G. Seemann, *AI for game developers* (O'Reilly, Sebastopol, CA, 2004).

[4] Clay Mathematics Institute, 2009. Millennium Problems, N vs NP. http://www.claymath.org/millennium/.

[5] C. Forgy, "Rete: a fast algorithm for the many pattern/many object pattern match problem", *Artificial Intelligence* **19** (1982) 17–37.

[6] M. I. Ginsberg, "Multivalued logics: a uniform approach to reasoning in artificial intelligence", *Comput. Intelligence* **4** (1988) 265–316.

https://doi.org/10.1017/S1446181109000352 Published online by Cambridge University Press

[7] R. Hahnle, "Advanced many-valued logic", in: *Handbook of philosophical logic*, 2nd ed, Volume 2 (eds D. M. Gabbay and F. Guenthner), (Kluwer Academic, Dordrecht, Holland, 2001) 297–395.
[8] A. A. Hopgood, *The state of artificial intelligence*, Volume 65 of *Advances in Computers* (ed. M. V. Zelkowitz), (Elsevier, Amsterdam, 2005) 1–75.
[9] S. N. Ramadas, A. Tweedie, L. O'Leary and G. Hayward, A rule based design toll for ultrasonic transdusers and arrays, Paper #1091, International Congress on Ultrasonics, Vienna, Apr. 9–13, 2007, Session R252007.
[10] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach* (Pearson Education, London, 2003).
[11] Stanford Encyclopedia of Philosophy, Paraconsistent. http://plato.stanford.edu/entries/logic-paraconsistent/.
[12] A. Stern, *Matrix logic* (North Holland, Amsterdam, 1988).
[13] A. Urquhart, "Basic many-valued logic", in: *Handbook of philosophical logic*, 2nd ed, Volume 2 (eds D. M. Gabbay and F. Guenthner), (Kluwer Academic, Dordrecht, 2001) 249–295.
[14] Wikipedia, 2009. Paraconsistent logic, http://en.wikipedia.org/wiki/Paraconsistent_logic.
[15] Wikipedia, 2009. NP-complete. http://en.wikipedia.org/wiki/NP-complete.
[16] R. G. Wolf, "A survey of many-valued logic (1966–1974)", in: *Modern uses of multiple-valued logic*, (eds J. M. Dunn and G. Epstein), (D Reidel Publishing, Dordrecht, 1975) 167–323.