SIMPLIFICATION OF TELEO-REACTIVE SEQUENCES

Seyed R. Mousavi, Krysia Broda
Computing Department, Imperial College
180 Queen's Gate, London SW7 2BZ,UK
{bsm99,kb}@doc.ic.ac.uk

Abstract

A Teleo-Reactive (TR) sequence is a sequence of <situation→ action> rules. The output of a TR sequence is the action part of the first rule in the sequence whose situation part evaluates to true. In this report, we present an algorithm for simplification of TR sequences, by which we mean to obtain another TR sequence, if any, that is smaller but semantically equal to the given one. The simplification algorithm can also be applied to decision lists, because a decision list is a special case of a TR sequence in that the only actions are true (1) and false (0). We also discuss how the algorithm can be extended in order to simplify multivariable decision trees. Finally, we extend the use of the simplification algorithm to simplifying classification rules.

Keywords*: TR programs, TR sequences, Ordered rules, Simplification, Classification rules, Rule induction, Data mining.*

1. Introduction

A *Teleo-Reactive (TR) sequence* is a sequence of $<s \rightarrow a>$ rules, where *s* denotes a conjunction of binary literals, and *a* denotes an action [1,2]:

$s_1 \rightarrow a_1$
$s_2 \rightarrow a_2$
…
$s_n \rightarrow a_n$

TR sequences were introduced in the context of robot control programs to provide easily-understood and robust control programs for robots involved in dynamic and unpredictable environments. The following example gives an informal idea about what a TR sequence is and how it works. A formal definition of TR sequences will be presented in the next section.

Example 1. Suppose there are a ball and a robot that can perform three actions: *rotate*, *move-forward*, and *kick* (the ball). Let these actions be denoted, respectively, by *r*, *m*, and *k*. Also suppose the robot can perceive, using its sensors, if it is *facing the ball*, denoted by *f*, and if it is *at the ball*, i.e. ready to kick it, denoted by *a*. The task that the robot is going to perform is to kick the ball. Table 1 shows two possible TR sequences, t_1 and t_2, that can be used as the control program in the robot to complete the task.

Table 1

t_1	t_2
$a \rightarrow k$	$a \rightarrow k$
$\neg a \wedge f \rightarrow m$	$f \rightarrow m$
$\neg a \wedge \neg f \rightarrow r$	$T \rightarrow r$

The robot uses such a TR sequence as follows. Continually, it evaluates the situations based on what it perceives using its sensors. The robot will take the action that corresponds to the *First True Situation (FTS)*, from the top. It will continue performing the action until the FTS changes. So, at anytime, the robot will be performing the action that corresponds to the current FTS.

Let us see how t_1 or t_2 can be used to complete the task. The first rule, either in t_1 or in t_2, tells the robot to kick the ball if at it. The second rule makes the robot perform the "move forward" action if the robot is not at the ball but is facing it. And the third rule makes the robot rotate if the robot is neither at the ball nor facing it.

Fig. 1 shows a possible scenario. First the robot is neither at the ball nor facing it. So the third rule is fired. Therefore, the robot rotates which will eventually make it -in normal conditions- face the ball, resulting in making the situation of the second rule the FTS. Then, the robot will move forward toward the ball while facing it. Therefore, in normal conditions, it will get to the ball meaning that the situation of the first rule will be the next FTS. So the robot will eventually kick the ball.

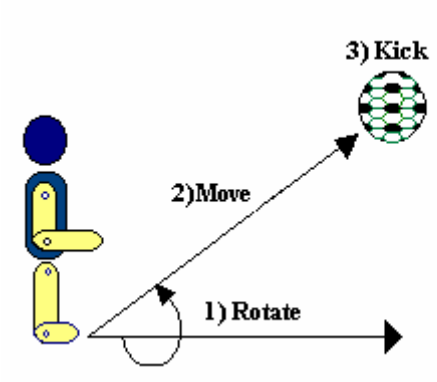


Fig. 1.

Note that both t_1 and t_2 suggest exactly the same actions at anytime, i.e. they are *equivalent*. However t_2 is more compact than t_1 in that it consists of fewer literals, resulting in less memory and probably higher speed in evaluating the situations. This illustrates the potential benefit of *simplifying* TR sequences, i.e. to convert them to more compact and still equivalent versions.

Although TR sequences were initially introduced to provide robot control programs, they can also be used as classification rules [3-5]. The main issue of using a TR sequence instead of a set of classification rules is that the rules would be order-dependent, i.e. they must be scanned sequentially from top to bottom. So, a TR sequence provides a *sequence*, rather than a *set*, of classification rules, and it may also be called *ordered* classification rules. Note that a set of classification rules is a special case of a TR sequence in which the situations are disjoint.

As an example, consider the following set of classification rules:

$a \rightarrow X$
$\neg a \wedge b \rightarrow Y$
$\neg a \wedge \neg b \wedge c \rightarrow Z$
$\neg a \wedge \neg b \wedge \neg c \rightarrow W$

Clearly, the rules are order-independent. The following TR sequence can replace this set of classification rules:

$a \rightarrow X$
$b \rightarrow Y$
$c \rightarrow Z$
$T \rightarrow W$

The advantage of using a TR sequence instead of order-independent classification rules is that the rules would normally be more compact and possibly more readable. However, the disadvantage is that the rules must be scanned from top to bottom. In other words, it is not possible to refer to a rule in the middle without checking the rules at the above of it. So, the use of a TR sequence as classification rules could be beneficial at-least in the applications where the top-to-bottom scan of the rules does not impose any additional cost.

Although producing smaller classification rules have been considered in the literature, taking the advantage of the ordering of the rules to do so has not been addressed in the literature, which is what this report presents. In this report, we present two algorithms, Remove-Rules and Remove-Literals, to simplify a given TR sequence. They remove, respectively, redundant rules and redundant literals, if any, from the given TR sequence. The main simplification algorithm will be a combination of these algorithms. As a set of classification rules is also a TR sequence, the simplification algorithm can be used in order to convert a set of classification rules to a TR sequence.

The rest of this report is organised as follows. In Section 2, we provide the basic definitions and define the problem. Section 3 presents a simplification algorithm to remove redundant rules from a TR sequence. In section 4, another algorithm is presented that removes redundant literals. The main simplification algorithm is provided in section 5. Section 6 discusses that the algorithms can be used for simplification of decision lists and that they can also be extended in order to simplify multivariable decision trees. Section 7 extends the use of the simplifications algorithms to simplifying classification rules. Experimental results are reported in Section 8; Section 9 concludes the report.

2. Basic Definitions

Definition 1. A *situation* is a conjunction of Binary literals.

Two special situations are the constant atoms denoted by *T* and *F* that, respectively, always evaluate to true and false. In this report, we use ".", in addition to $\wedge$, "+", in addition to $\vee$, and "′ " to denote, respectively, conjunction, disjunction, and negation.

Definition 2. Let $\mathsf{A} = \{a_1, a_2, \ldots a_n\}$ denote a set of possible classes (or actions). A *Teleo-Reactive (TR) rule*, or simply a *rule*, is a pair of $<s,a>$, where *s* is a situation and *a* is an element in A.

In this report, we use *r* to denote a rule and $s \rightarrow a$ instead of $<s,a>$.

Definition 3. A *TR sequence* is a sequence of rules.

So if *t* is a TR sequence, $t=<r_1, r_2,\ldots,r_n>$, where r_i denotes the i^{th} rule in the sequence. The most common way to represent such a TR sequence, however, is the following:

$s_1 \rightarrow a_1$
$s_2 \rightarrow a_2$
…
$s_n \rightarrow a_n$

where $s_i \rightarrow a_i$ denotes the i^{th} rule, r_i.

Definition 4. Let $t=<r_1,r_2,\ldots,r_n>$ be a TR sequence. *t* is called *complete* if the disjunction of all the situations in it is a tautology, i.e. $\bigvee_{k=1}^{n} s_k \equiv T$.

Example 2. t_1 and t_2 in Table 1 are both complete TR sequences because:

For t_1: $a + a' + f + a' + f' \equiv T$, and for t_2: $a + f + T \equiv T$

Definition 5. Let $t=<r_1,r_2,\ldots,r_n>$ be a TR sequence. The *First True Situation* (*FTS*) in t is the first situation, i.e. the situation with the lowest index, that evaluates to true; that is $s_k=FTS$ iff ($s_k=T$) and $\forall\, s_j, j<k \Rightarrow s_j=F$.

Note that "$=$" here means "evaluates to" rather then "is equivalent to", which is denoted by "$\equiv$".

Definition 6. Let t be a TR sequence. By the *output of t*, denoted by $t(\,)$, we mean the action part of the rule whose situation is the FTS, that is $t(\,)=a_k$ iff s_k=FTS.

Clearly, if t is complete then the output of t will always be defined, because the FTS always exists. For this reason, a complete TR sequence may be used in a robot as the control program, i.e. to determine the action to take at a given time. However, if the TR sequence is not complete, no action can be determined when none of the situations evaluates to true.

Example 3. Consider t_2 in Table 1 and the scenario shown in Fig. 1. Since the robot is neither at the ball nor facing it, only the third situation, T, evaluates to true; so FTS=T and $t_2(\,)$="rotate". And performing this action, in normal conditions, makes the robot face the ball, in which case the FTS and the output of the TR sequence will be, respectively, f and "move-forward". Similarly, performing "move-forward" will, in normal conditions, make the robot get to the ball, which means the next FTS will be a, and the robot will then kick the ball.

Definition 7. Let t_1 and t_2 be TR sequences. We say t_1 and t_2 are *equivalent*, and write $t_1 \eqqcolon t_2$ or $t_2 \eqqcolon t_1$ if, at anytime, having the same values for the atoms used in the situations, the output of t_1 is the same as the output of t_2, i.e. either both of the outputs are defined and are the same or neither of them is defined.

Definition 8. Let s be a situation in a TR sequence. The *length* of s, denoted by $l(s)$, is the number of literals in s. The length of T and F are both defined as 0.

Definition 9. Let t be a TR sequence. The length of t is defined as $l(t)=\sum_{s \text{ is a situation in } t} l(s)$.

The simplification problem. Let t be a TR sequence. By simplifying t we mean to obtain another TR sequence, say t_s, if any, such that $t_s \eqqcolon t$ and $l(t_s)< l(t)$. Ideally, t_s would be the most simplified form of the TR sequence, which means it cannot be simplified anymore, i.e. for every t_1, $(t_1 \eqqcolon t) \Rightarrow\ l(t_s) \leq l(t_1)$. The algorithms we present in this report, however, do not guarantee to provide the most simplified form of a given TR sequence.

The purpose of simplifying a TR sequence is to have a smaller one that still is semantically equal to the given TR sequence. It can result in less required storage and possibly more readable TR sequence. In the case of robot control, it may also result in faster processing and therefore faster responses to input stimuli.

Example 4. t_1 and t_2 in Table 1 are equivalent, but t_1 is smaller than t_2 because:

$$l(t_1)=l(a)+ l(a′.f) + l(f)=4 \text{ and } l(t_2)=l(a)+ l(f) + l(T)=2$$

So t_2 is a simplified version of t_1.

During the rest of the report, we assume that the input TR sequence is the following:

$s_1 \rightarrow a_1$
$s_2 \rightarrow a_2$
…
$s_n \rightarrow a_n$

3. Removing Redundant Rules

In this section, we provide an algorithm that removes all the redundant rules, if any, from a given TR sequence. A special case of redundant rules is a never-executed rule. We first present an algorithm to remove never-executed rules and then extend it to remove any redundant rules.

Definition 10. A *never-executed* rule is a rule whose situation will never be the FTS.

Example 5. Consider TR sequence t_1 in Table 2.a. The second rule, $a.b.d \rightarrow a_2$, is a never executed rule because if its situation evaluates to true, the situation of the first rule will also evaluate to true, i.e. $a.b.d \Rightarrow a.d$, and therefore the situation of the second rule can never be the FTS.

The fourth rule, $b.c.d \rightarrow a_4$, is also a never-executed rule because if its situation evaluates to true, at least one of the situations of the third or the first rules will also evaluate to true and therefore the situation of the fourth rule can never be the FTS. That is because $b.c.d \Rightarrow a'.c.d + a.d$. So t_2 shown in Table 2.b will be a simplified version of t_1.

Table 2.

(a) t_1	(b) t_2	(c) t_3	(d) t_4
$a.d \rightarrow a_1$ $a.b.d \rightarrow a_2$ $a'.c.d \rightarrow a_3$ $b.c.d \rightarrow a_4$ $d'.e \rightarrow a_5$ $c \rightarrow a_3$ $c'.d.e \rightarrow a_6$	$a.d \rightarrow a_1$ $a'.c.d \rightarrow a_3$ $d'.e \rightarrow a_5$ $c \rightarrow a_3$ $c'.d.e \rightarrow a_6$	$a.d \rightarrow a_1$ $d'.e \rightarrow a_5$ $c \rightarrow a_3$ $c'.d.e \rightarrow a_6$	$a.d \rightarrow a_1$ $d'.e \rightarrow a_5$ $c \rightarrow a_3$ $e \rightarrow a_6$

A never-executed rule is in fact a rule whose situation will not be true unless the situation of another rule of a higher order, i.e. a lower index, is true, i.e.:

r_i $(i>1)$ is a never-executed rule iff $s_i \Rightarrow \bigvee_{1 \le j < i} s_j$

Therefore, the following algorithm can be used in order to remove never-executed rules:

Algorithm Remove_never-executed_rules:

For all rules, r_i, from r_1 to r_n do
Build $L_{above,i} = \bigvee_{1 \le j < i} s_j$
If $s_i \Rightarrow L_{above,i}$ *then* //*if* s_i *can never be the* FTS
Delete r_i
End For
End

Theorem. (a) If r_k is a rule that algorithm Remove_never-executed_rules deletes, r_k is a never-executed rule. ***(b)*** If r_k is a never-executed rule, then the algorithm will delete it.

Proof.
(a). The algorithm deletes rule r_k only when the *If* condition evaluates to true. So if r_k is removed by the algorithm, then $s_k \Rightarrow L_{above,k}$, which means $s_k \Rightarrow \bigvee_{1 \le j < k} s_j$, that is r_k is a never-executed rule.

(b). It is clear that if r_k is a never-executed rule at the time $i=k$ in the algorithm, then the algorithm will remove it. So we only need to see whether r_k will still be a never-executed rule if a higher rule is

removed. Let assume that r_m, $m<k$ was a never-executed rule and removed. We would like to see if r_k is still a never-executed rule. In other words, we have:

$$s_m \Rightarrow \bigvee_{1\le j<m} s_j$$

$$s_k \Rightarrow \bigvee_{1\le j<k} s_j$$

and we would like to see if $s_k \Rightarrow \bigvee_{\substack{1\le j<i \\ j\ne m}} s_j$

$$(s_k \Rightarrow \bigvee_{1\le j<k} s_j)$$

$$\equiv (s_k \Rightarrow \bigvee_{\substack{1\le j<i \\ j\ne m}} s_j \vee s_m)$$

Hence $(s_k \Rightarrow \bigvee_{\substack{1\le j<i \\ j\ne m}} s_j \vee (\bigvee_{1\le j<m} s_j))$, because we have $s_m \Rightarrow \bigvee_{1\le j<m} s_j$

$$\equiv (s_k \Rightarrow \bigvee_{\substack{1\le j<i \\ j\ne m}} s_j)$$

So, if s_k is a never-executed rule, it will still be so even if a higher never executed rule is removed.

In addition to never-executed rules, there might be other rules that are also redundant. Such rules may fire in some circumstances as they are not never-executed rules. However, in a such a circumstance, another rule with the same action will be fired if the rule is removed from the TR sequence. Now, we generalise the discussion and derive an algorithm that removes any redundant rules.

Definition 11. A *redundant* rule is a rule whose removal from the underlined TR sequence does not affect the output of the TR sequence.

Example 6. The second rule in TR sequence t_2 (Table 2.b) is a redundant rule although it is not a never-executed rule. To see why it is redundant, remove it from the TR sequence. Then t_3 (Table 2.c) will result. That rule could be fired, in t_2, only if $a=F$, $c=T$, and $d=T$, in which case the second rule in t_3 would fail and the fourth rule would be fired resulting in taking the same action, a_3, as the action of the removed rule. This means that even if the rule is removed from the TR sequence, the output of the TR sequence remains the same.

Now, let us see when a rule is redundant. Let $t=<r_1,\ldots r_{k-1},r_k,r_{k+1},\ldots,r_n>$ be a TR sequence, where r_i denote $s_i \rightarrow a_i$. Then the TR sequence resulting from removing r_k from t will be $t_S=<r_1,\ldots r_{k-1},r_{k+1},\ldots,r_n>$. Obviously r_k in t is a redundant rule iff $t \equiv t_s$, which means (r_k is redundant) $\equiv$ ($t( )=t_s( )$). It is clear that the right hand side of the equation will hold in the case that FTS in t is one the situations above s_k or one of the situations below it. So the only remaining case is when the FTS in t is s_k. In other words, r_k is redundant in t if and only if (s_k=FTS $\Rightarrow$ $t( )= t_s( )$) or simply (s_k=FTS $\Rightarrow t_s( )=a_k$).

One circumstance that this condition holds is when s_k cannot be the FTS, i.e. when r_k is a never-executed rule, which was discussed before. Clearly, if s_k=FTS in t, then none of the situations s_i, $i<k$ can be the FTS in t_s. So the FTS in t_s must be a s_i such that $i>k$. therefore, we can write:

$$(r_k \text{ is redundant}) \equiv (s_k=\text{FTS in } t \Rightarrow \exists\, s_j, j>k, a_j=a_k, s_j=\text{FTS in } t_s)$$

$$\equiv ((\bigvee_{1\le i<k} s_j)' \wedge s_k) \Rightarrow \exists\, s_j, j>k, a_j=a_k, (\bigvee_{\substack{1\le m<j \\ m\ne k}} s_m)' \wedge s_j)$$

$$\equiv ((\bigvee_{1\le i<k} s_j)' \wedge s_k) \Rightarrow \exists\, s_j, j>k, a_j=a_k, ((\bigvee_{1\le m<k} s_m) \vee (\bigvee_{k<m<j} s_m))' \wedge s_j)$$

$$\equiv ((\bigvee_{1\le i<k} s_j)' \wedge s_k) \Rightarrow \exists\, s_j, j>k, a_j=a_k, (\bigvee_{1\le m<k} s_m)' \wedge (\bigvee_{k<m<j} s_m)' \wedge s_j)$$

Note r_k does not exist in t_s. Now, let us define:

$$L_{above,k}=\bigvee_{1\le i<k} s_j \quad \text{and} \quad L_{below,k,j}=(\bigvee_{k<m<j} s_m)' \wedge s_j)$$

Then we will have:

$$(r_k \text{ is redundant}) \equiv ((L_{above,k})' \wedge s_k) \Rightarrow \exists s_j, j>k, a_j=a_k, (L_{above,k})' \wedge L_{below,k,j}$$
$$\equiv ((L_{above,k})' \wedge s_k) \Rightarrow \exists s_j, j>k, a_j=a_k, L_{below,k,j}$$

Now, let us define $A_{below,k,a_k} = \bigvee_{\substack{j>k \\ a_j=a_k}} L_{below,k,j}$. Note that if there is no operand for the $\vee$ operator, we will assume that the expression evaluates to false. Then we will have:

$$(r_k \text{ is redundant}) \equiv ((L_{above,k})' \wedge s_k) \Rightarrow A_{below,k,a_k}$$
$$\equiv ((L_{above,k})' \wedge s_k)' \vee A_{below,k,a_k}$$
$$\equiv ((L_{above,k}) \vee s_k' \vee A_{below,k,a_k}$$

and finally

$$(r_k \text{ is redundant}) \equiv (s_k \Rightarrow L_{above,k} \vee A_{below,k,a_k}) \qquad (Eq.1)$$

Now, we present the algorithm that removes redundant rules, including never-executed rules:

***Algorithm Remove_Rules**:*

For all rules, r_i, from r_1 to r_n do

a. Build $L_{total,i}=L_{above,i} \vee A_{below,i,a_i}$

b. If $s_i \Rightarrow L_{total,i}$ then // if s_i is either a never-executed rule or not a never-executed rule but still redundant

Delete r_i

End For

End

Theorem. (a) If r_k is a rule that algorithm Remove_Rules deletes, r_k is a redundant rule. ***(b)*** If r_k is a redundant rule, then algorithm will delete it unless it is used to delete a higher redundant rule.

Proof.

(a). The algorithm deletes rule r_k only when the condition $s_i \Rightarrow L_{above,k} + A_{below,i,ak}$ evaluates to true, which means r_k is a redundant rule.

(b). It is clear that if r_k is a redundant rule when $i=k$ in the algorithm, then it will be removed. So we only need to see it will still be a redundant rule if a higher rule is removed, unless it is used to remove that rule. Let assume that r_m, $m<k$ was a redundant rule and removed. We have the following assumptions:

(1) $s_m \Rightarrow L_{above,m} \vee A_{below,m,a_m}$

(2) $s_k \Rightarrow L_{above,k} \vee A_{below,k,a_k}$

(3) (r_k has not been used in removing r_m)

and we would like to show that r_k will still be redundant after removing r_m, i.e.:

$$(s_k \Rightarrow \bigvee_{\substack{1\le j<i \\ j\neq m}} s_j \vee A_{below,k,a_k})$$

Based on assumption (1), we have:

$$(s_m \Rightarrow L_{above,m} \vee A_{below,m,a_m})$$
$$\equiv (s_m \Rightarrow \bigvee_{1\le j<m} s_j \vee (\exists r_q, m<q, a_q=a_m, L_{below,m,q})$$

Because of assumption (3), q cannot be equal to k, and the above expression will be equivalent to:

$$(s_m \Rightarrow \bigvee_{1\le j<m} s_j \vee (\exists r_q, (m<q<k \text{ or } k<q), a_q=a_m, L_{below,m,q}) \qquad (4)$$

On the other hand we have:

Assumption (2) implies $(s_k \Rightarrow \underset{\substack{1\le j<k \\ j\neq m}}{\vee} s_j \vee s_m \vee A_{below,k,a_k})$

Then using (4):

Assumption (2) implies $(s_k \Rightarrow \underset{\substack{1\le j<k \\ j\neq m}}{\vee} s_j \vee \underset{1\le j<m}{\vee} s_j \vee (\exists r_q, (m<q<k \text{ or } k<q), L_{below,m,q}) \vee A_{below,k,a_k})$

Which implies $(s_k \Rightarrow (\underset{\substack{1\le j<k \\ j\neq m}}{\vee} s_j) \vee A_{below,k,a_k} \vee (\exists r_q, (m<q<k \text{ or } k<q), L_{below,m,q})$ *(5)*

Now, let us define:

$X=(\exists r_q, m<q<k, a_q=a_m, L_{below,m,q})$
$Y=(\exists r_q, k<q, a_q=a_m, L_{below,m,q})$

Then we have:

$$X \Rightarrow \exists r_q, m<q<k, q_q=a_m, (\underset{q<j<m}{\vee} s_j)' \wedge s_q)$$
$$\Rightarrow \exists r_q, q<k, s_q$$
$$\Rightarrow (\underset{\substack{1\le j<k \\ j\neq m}}{\vee} s_j)$$

and

$$Y \Rightarrow \exists r_q, k<q<n, q_q=a_m, (\underset{q<j<m}{\vee} s_j)' \wedge s_q)$$
$$\Rightarrow \exists r_q, k<q, (\underset{q<j<m}{\vee} s_j)')$$
$$\Rightarrow s_k'$$

Therefore:

$$(5) \Rightarrow (s_k \Rightarrow (\underset{\substack{1\le j<i \\ j\neq m}}{\vee} s_j) \vee A_{below,k,a_k} \vee X \vee Y)$$
$$\Rightarrow (s_k \Rightarrow (\underset{\substack{1\le j<i \\ j\neq m}}{\vee} s_j) \vee A_{below,k,a_k} \vee (\underset{\substack{1\le j<k \\ j\neq m}}{\vee} s_j) \vee s_k')$$
$$\Rightarrow (s_k \Rightarrow \underset{\substack{1\le j<i \\ j\neq m}}{\vee} s_j \vee A_{below,k,a_k})$$

and the proof is complete. So, if r_k has not been used in removing r_m, then it will still be redundant after removal of r_m.

4. Removing Redundant Literals

Redundant literals in a given TR sequence are literals whose removal does not affect the output of the TR sequence.

Example 7. Consider t_3 in Table 2.c. Literal c in the last rule, $c'.d.e \rightarrow a_6$, is redundant because if neither of its above situations is FTS, then c will certainly be false, because if c is true and the first and the second situations are not FTS, then the third situation will be the FTS. Literal d in the last rule is also redundant, because it can never be false unless $e=F$ (otherwise the second rule would be fired) in which case the rule would not fire. So whether or not it is in the situation does not affect when the rule is fired. Therefore t_3 is equivalent to t_4 shown in Table 2.d.

Now, let us see when a literal is redundant. Let $t=<r_1,\ldots,r_k,,\ldots,r_n>$ be a TR sequence where r_i denote $s_i \rightarrow a_i$, and $s_k= \bigwedge_{r=1}^{p} l_r$ where l_r is a literal. Then let t_s denote the TR sequence resulted by removing l_d from s_k. $t_=$ is exactly the same as t except that $s_k= \bigwedge_{r=1}^{p} l_r$ in t but $s_k= \bigwedge_{\substack{r=1 \\ p\neq d}}^{p} l_r$ in t_s. Obviously l_d is a redundant literal iff $t=t_s$, which means (l_d is redundant) $\equiv$ *($t( )=t_S( )$)* It is clear that the right hand side of the equation will hold when FTS in t_s is one the situations above s_k or one of the situations below it. So the only remaining case is when the FTS in t_s is s_k. In other words, r_k is redundant in t if and only if *(s_k=FTS in $t_s \Rightarrow$ $t( )=t_s( )$)* or simply *(s_k=FTS in $t_s \Rightarrow t_s( )=a_k$.*.

Obviously, one circumstance that this condition holds is when s_k cannot be FTS, which is what discussed under never-executed rules. So here we assume that s_k can be the FTS in t, and we would like to see under which conditions the condition holds. If s_k=FTS in t_S, then none of the situations s_i, $i<k$ can be the FTS in t. So the FTS in t must be an s_i such that $i \geq k$. So, we have:

(l_d is redundant) $\equiv$ *(s_k=FTS in $t_s \Rightarrow$ s_k=FTS in t) $\vee$ ($\exists s_j, j>k, a_j=a_k, s_j$=FTS in t_s)*
$\equiv$ *(s_k=FTS in $t_s \Rightarrow$ s_k=FTS in t) $\vee$ (s_k=FTS in $t_s \Rightarrow \exists s_j, j>k, a_j=a_k, s_j$=FTS in t)*
(Eq.II)

The left operand of the $\vee$ operator in this equation is equivalent to the following:

(s_k=FTS in $t_S \Rightarrow$ s_k=FTS in t)

$$\equiv ((\bigvee_{1\leq i<k} s_j)' \wedge \bigwedge_{\substack{r=1 \\ p\neq d}}^{p} l_r) \Rightarrow (\bigvee_{1\leq i<k} s_j)' \wedge \bigwedge_{r=1}^{p} l_r$$

$$\equiv ((L_{above,k})' \wedge \bigwedge_{\substack{r=1 \\ p\neq d}}^{p} l_r) \Rightarrow (L_{above,k})' \wedge \bigwedge_{\substack{r=1 \\ p\neq d}}^{p} l_r \wedge l_d$$

Now, let us define $L_{literals,k,l_d} = \bigwedge_{\substack{l \text{ is a literal in } s_k \\ l\neq l_d}} l$. Then this expression will be equivalent to:

$$(L_{above,k})' \wedge L_{literals,k,l_d} \Rightarrow (L_{above,k})' \wedge L_{literals,k,ld} \wedge l_d$$

$$\equiv (L_{above,k})' \wedge L_{literals,k,l_d} \Rightarrow l_d$$
$$\equiv L_{above,k} \vee (L_{literals,k,l_d})' \vee l_d$$

On the other hand, the right operand of the $\vee$ operator in *Eq. II* will similarly be simplified:

(s_k=FTS in $t_s \Rightarrow \exists s_j, j>k, a_j=a_k, s_j$=FTS in t)

$$\equiv ((\bigvee_{1\leq i<k} s_j)' \wedge l1\ldots.l_{d-1}.l_{d+1}\ldots.lp \Rightarrow \exists s_j, j>k, a_j=a_k, (\bigvee_{1\leq m<j} s_m)' \wedge s_j)$$

$$\equiv (L_{above,k})' \wedge L_{literals,k,l_d} \Rightarrow \exists s_j, j>k, a_j=a_k, ((\bigvee_{1\leq m<k} s_m) \vee (\bigvee_{k<m<j} s_m))' \wedge s_j)$$

$$\equiv (L_{above,k})' \wedge L_{literals,k,l_d} \Rightarrow \exists s_j, j>k, a_j=a_k, (\bigvee_{1\leq m<k} s_m)' \wedge (\bigvee_{k<m<j} s_m)' \wedge s_j)$$

$$\equiv (L_{above,k})' \wedge L_{literals,k,l_d} \Rightarrow \exists s_j, j>k, a_j=a_k, (L_{above,k})' \wedge L_{below,k,j}$$

$$\equiv (L_{above,k})' \wedge L_{literals,k,l_d} \Rightarrow \exists s_j, j>k, a_j=a_k, L_{below,k,j}$$

$$\equiv (L_{above,k})' \wedge L_{literals,k,l_d} \Rightarrow A_{below,k,ak}$$
$$\equiv L_{above,k} \vee (L_{literals,k,l_d})' \vee A_{below,k,ak}$$

So l_d is redundant $\equiv (L_{above,k} \vee (L_{literals,k,l_d})' \vee l_d) \vee (L_{above,k} \vee (L_{literals,k,l_d})' \vee A_{below,k,a_k})$
$\equiv L_{above,k} \vee (L_{literals,k,l_d})' \vee l_d \vee A_{below,k,a_k}$
$\equiv (l_d' \Rightarrow L_{above,k} \vee (L_{literals,k,l_d})' \vee A_{below,k,a_k})$ *(Eq. III)*

Now, we present the algorithm that removes redundant literals, including never-executed rules:

***Algorithm Remove_Literals**:*
For all the rules, r_i, from r_1 to r_n do
a. Build $L_{above,i}$ and $A_{below,i,ai}$

b. For each literal l in s_i do
Build $L_{literals,i,l}$
Build $L_{total,i,l} = L_{above,i} + (L_{literals,i,l})' + A_{below,i,a_i}$
If $l' \Rightarrow L_{total,i,l}$ then remove l from r_i
End For
End For
End

Theorem 3. Let l_d be a literal in situation s_k. ***(a)*** If algorithm Remove_Literal deletes l_d *from* s_k, then l_d is a redundant literal. ***(b)*** If l_d is a redundant literal, then the algorithm will delete it from s_k.

Proof.
(a). The algorithm deletes literal l_d only when the condition of *If*, i.e. $l_d' \Rightarrow L_{total,k,l_d}$, evaluates to true. Therefore if l_d is removed by the algorithm, we will have:

$$(l_d' \Rightarrow L_{total,k,ld})$$
$$\equiv (l_d' \Rightarrow L_{above,k} + (L_{literals,k,l_d})' + A_{below,k,a_k})$$
$$\equiv (l_d \text{ is redundant}), \text{ because of } Eq.\ III.$$

(b). It is clear that if l_d is a redundant literal when $i=k$ in the algorithm, then it will be removed. So we only need to see whether it will still be a redundant literal if a literal in a higher rule is removed. Let assume that literal l_c in r_m, $m<k$ was a redundant literal and removed. We have:

$$l_d' \Rightarrow L_{above,k} + (L_{literals,k,l_d})' + A_{below,k,a_k}$$

Clearly, removing l_c from s_m will only affect $L_{above,k}$ and does not change $L_{literals,k,ld}$ and A_{below,k,a_k}. Let L^{before} and L^{after} denote respectively $L_{above,k}$ before and after the removal of l_c from s_m. Then we have:

$$l_d' \Rightarrow L^{before} + (L_{literals,k,l_d})' + A_{below,k,a_k}$$

Also, let s_m^{before} and s_m^{after} denote respectively s_m before and after the removal of l_c. Then we will have:

$$l_d' \Rightarrow (\bigvee_{\substack{1\le j<k \\ j\neq m}} s_j \vee s_m^{before}) + (L_{literals,k,l_d})' + A_{below,k,a_k}$$

On the other have we have:

$$s_m^{before} \equiv s_m^{after} \wedge l_c.$$

So, we will have:

$$(l_d' \Rightarrow \bigvee_{\substack{1\le j<k \\ j\neq m}} s_j \vee (s_m^{after} \wedge l_c) + (L_{literals,k,l_d})' + A_{below,k,a_k})$$
$$\Rightarrow (l_d' \Rightarrow \bigvee_{\substack{1\le j<k \\ j\neq m}} s_j \vee (s_m^{after}) + (L_{literals,k,l_d})'_d + A_{below,k,a_k})$$
$$\Rightarrow (l_d' \Rightarrow L^{after} + (L_{literals,k,l_d})' + A_{below,k,a_k})$$

which means that literal l_d is still a redundant literal in s_k after the removal of l_c.

Example 8. Consider Example 1. Simplifying t_1 will result in t_2.

Example 9. Consider t_8 and t_9 defined in Table. 3. Simplifying t_8 will result in t_9.

Table 3

t_8	t_9
$a.b' \rightarrow a_1$ $a'.b \rightarrow a_2$ $a \rightarrow a_1$ $a'.b'.c \rightarrow a_3$ $b.c \rightarrow a_5$ $a.d' \rightarrow a_3$ $a'.b'.c' \rightarrow a_4$	$a \rightarrow a_1$ $b \rightarrow a_2$ $c \rightarrow a_3$ $T \rightarrow a_4$

For instance, consider rule $b.c \rightarrow a_5$ in t_8. This rule is a never-executed rule, because if $b.c=T$ then one of the higher situations will be true. That is because $b.c \Rightarrow a'.b'.c + a + a'.b + a.b'$. So this rule does not exist in t_9. Now let us see how Algorithm Remove_Rules removes this rule from t_8. When i=5 in the loop, the algorithm proceeds as follows: In step (a) it generates the following sets:

$$L_{above,5} = \bigvee_{1 \le j < 5} s_j = a.b' + a'.b + a + a'.b'.c = a + b + c$$

$$A_{below,5,a_5} = \bigvee_{\substack{j>5 \\ a_j=5}} L_{below,j} = F$$

$$L_{total,5} = L_{above,5} + A_{below,5,a_5} = a + b + c$$

Then, in step (b) of the algorithm $s_5 \Rightarrow L_{total,5}$ evaluates to *true* because*:*

$$(s_5 \Rightarrow L_{total,5}) \equiv (b.c \Rightarrow a + b + c) \equiv T.$$

So r_5, $b.c \rightarrow 5$, is removed from the TR sequence.

Now, consider the first rule in t_8. Literal b' is redundant because even if b is true, i.e. $a.b=T$, the same action, a_1, will be the output, because in this case the second rule will fail and the third rule will be fired. The algorithm, *Remove_Literals*, removes the literal as follows: When i=1 in the outer loop, it generates the following sets during step (a):

$$L_{above,1} = \bigvee_{1 \le j < 5} s_j = F$$

$$A_{below,1,a_1} = \bigvee_{\substack{j>1 \\ a_j=1}} L_{below,1,j=} L_{below,1,3=} (\bigvee_{1<m<3} s_m)' \wedge s_3 = (s_2)' \wedge s_{3=} (a'.b)' \wedge a = a$$

Then, in part (b), when $l=b'$, $(b')' \Rightarrow L_{above,1}$ evaluates to false, and the following are generated, in the *else* branch of the *if* condition:

$$L_{literals,1,l=} a$$
$$L_{total,1,l} = L_{above,1} + (L_{literals,1,l})' + A_{below,1,a_1} = F + a' + a = T$$

Next, $b \Rightarrow L_{total,1,l}$ evaluates to T, and finally b' is removed from r_1.

5. The Main Simplification Algorithm

The main simplification algorithm, Remove_Rules_and_Literals, is an algorithm that combines the above algorithms, Remove_Rules and Remove_Literals. The reason why it is not simply a call of Remove_Rules followed by a call of Remove_Literals is that removing literals can result in making a rule redundant. On the other hand, it cannot be a call of Remove_Literals followed by a call of

Remove_Rules because removing a literal from a never-executed rule can make it a non-redundant rule. So, we have used a special combination of the two algorithms to devise algorithm Remove_Rules_and_Literals.

***Algorithm Remove_Rules_and_Literals**:*

For all rules, r_i, from r_1 to r_n do

a. Build $L_{above,i}$, and A_{below,i,a_i}

b. If $s_i \Rightarrow L_{above,i} + A_{below,i,a_i}$ then // if r_i is a redundant rule

Delete r_i

Else

. For each literal l in s_i do

Build $L_{literals,i,l}$

If $l' \Rightarrow L_{above,i} + (L_{literals,i,l})' + A_{below,i,a_i}$ then remove l from r_i

End For

End If

End For

End

Theorem 3. (a) If r_k is a rule that algorithm Remove_Rules_and_Literals deletes, r_k will be a redundant rule. ***(b)*** If r_k is a redundant rule then the algorithm will delete it unless it is used to delete a higher redundant rule. ***(c)*** Let l_d be a literal in situation s_k. Then, if the algorithm deletes l_d from s_k, then l_d is a redundant literal.

Proof.

(a). The algorithm deletes rule r_k only when the condition $s_i \Rightarrow L_{above,k} + A_{below,i,ak}$ evaluates to true, which means r_k is a redundant rule.

(b). It is clear that if r_k is a redundant rule when $i=k$ in the algorithm, then it will be removed. So we only need to see it will still be a redundant rule if (1) a higher rule is removed, unless it is used to remove the rule or (2) a literal in a higher rule is removed. Case (1) has already been shown in theorem2; so we here prove case (2).

Let assume that literal l_c in r_m, $m<k$ was a redundant literal and removed. We would like to see if r_k is still redundant. Clearly, removing l_c from s_m will only affect $L_{above,k}$ and does not change A_{below,k,a_k}. Let L^{before} and L^{after} denote respectively $L_{above,k}$ before and after the removal of l_c from s_m. Then, since rk is redundant before removing lc, we have:

$$s_k \Rightarrow L^{before} \vee A_{below,k,a_k}$$

Also, let s_m^{before} and s_m^{after} denote respectively s_m before and after the removal of l_c. Then we will have:

$$s_k \Rightarrow (\bigvee_{\substack{1 \le j<k \\ j \ne m}} s_j \vee s_m^{before}) \vee A_{below,k,a_k}$$

On the other have we have:

$$s_m^{before} \equiv s_m^{after} \wedge l_c.$$

So, we will have:

$$s_k \Rightarrow \bigvee_{\substack{1 \le j<k \\ j \ne m}} s_j \vee (s_m^{after} \wedge l_c) \vee A_{below,k,a_k}$$

$$\Rightarrow (s_k \Rightarrow \bigvee_{\substack{1 \le j<k \\ j \ne m}} s_j \vee (s_m^{after}) \vee A_{below,k,a_k})$$

$$\Rightarrow (s_k \Rightarrow L^{after} \vee A_{below,k,a_k})$$

which means that r_k is still redundant, after the removal of l_c.

(c). The algorithm deletes literal l_d only in two cases: (1) when $(l_d' \Rightarrow L_{above,k})$ in part b-1. and (2) when $(l_d' \Rightarrow L_{total,k,ld})$ in part b-2. So if l_d is removed by the algorithm, then:

$(l_d' \Rightarrow L_{above,k}) \vee (l_d' \Rightarrow L_{total,k,ld})$
$\equiv l_d' \Rightarrow L_{above,k} \vee L_{total,,k,ld}$
$\equiv l_d' \Rightarrow L_{above,k} \vee (L_{literals,k,l_d})' \vee A_{below,k,a_k}$
$\equiv (l_d$ is redundant), because of *Eq. III*.

Note that if l_d is a redundant literal, we cannot say that the algorithm will delete it from s_k. That is because a literal may be redundant before the removal of a rule but not after it. For example consider the following TR sequence:

$a.b \rightarrow a_1$
$a'.b \rightarrow a_2$
$b \rightarrow a_1$

The first rule is redundant because if *a* and *b* are both true then the second rule will fail and the third rule will fire resulting in the same action. If the first rule is not removed, literal a' in the second rule will be redundant. But if it is removed, the literal will no longer be redundant. So, this algorithm prioritises deletion of redundant rules over deletion of redundant literals.

6. Simplifying Decision Lists and Multivariable Decision Trees

A decision list is a list of (f_i,v_i) pairs where f_i is a conjunction of literals and v_i is either true (1) or false (0) [6]:

(f_1,v_1)
…
(f_n,v_n)

f_n, i.e. that last f_i, is the constant Boolean function *T* that is always true. A decision list *L* defines a boolean function *L(X)* where *X* is the input vector. For any input *X*, *L(X)* is defined to be equal to v_i where *i* is the least index such that $f_i(X)=T$. A decision list may be thought of as an extended "if-then-elseif-…else- " instruction. We borrow the following example from [6]:

Example 10. Consider the following decision list *L*:

$(x_1.x_3',0)$
$(x_1'.x_2.x_5,1)$
$(x_3'.x_4',1)$
$(T,0)$

L defines a Boolean function over binary variable x_1,x_2,x_3,x_4, and x_5. Fig. 2 shows the Karnaugh map for *L*. In fact, *L* is equivalent to binary function $f= x_1'.x_2.x_5 + x_1'.x_3'.x_4'$.

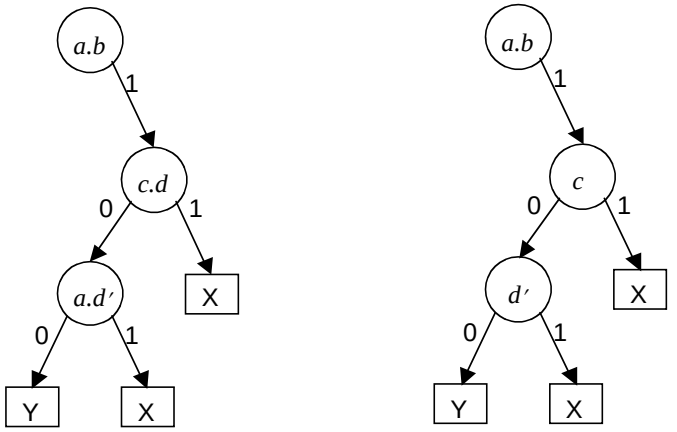
x_1x_2 \ $x_3x_4x_5$	000	001	011	010	100	101	111	110
00	1	1	0	0	0	0	0	0
01	1	1	1	0	0	1	1	0
11	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0

Fig. 2.

Obviously, a decision list can be viewed of as a special type of a TR sequence in that the only actions are "0" and "1" [2]. Therefore, the presented simplification algorithms are applicable to decision lists as well. Consider, for instance, decision list L in example 10. Algorithm Remove-Rules does not affect L as there is no redundant rule in it, but algorithm Remove_Literals removes literal x_3' from the first rule and then x' from the second rule resulting in the following decision list:

$(x_1,0)$
$(x_2,x_5,1)$
$(x_3'.x_4',1)$
$(T,0)$

On the other hand, a TR sequence is a special type of a binary multivariable decision tree. In such a decision tree, each node corresponds to a conjunction of some binary literals, and therefore evaluates to either true or false. The simplification algorithms, therefore, can be viewed of as special cases of more general algorithms that can be used to simplify general binary multivariable decision trees. Fig. 3 shows, as an example, a possible sub-tree of a binary multivariable decision tree and its simplified version.



(a) before simplification (b) after simplification

Fig. 3.

However, it is important to note that there will be no need for simplification if the algorithm used to construct such a tree has already avoided such redundancy, which is quite possible.

7. Simplifying Classification Rules

In this section, we focus on the case where we are given a set of order-independent classification rules, rather than a TR sequence, which we would like to simplify. We assume that each attribute is discrete and has a finite set of nominal values, e.g. {1, 2, 3}. Such a set of rules could be the output of a rule induction algorithm such as ID3 [7]. We would like to extend the use of the simplification algorithm to simplifying such rules.

The first issue in applying the simplification algorithm to classification rules is that the resulting rules will no longer be order-independent; i.e. it is not possible to interpret a rule in the middle independently from the rules above. It is nevertheless possible to retain some degree of order independence in the output rules by the following means: sort the input rules by their classes so that all the rules having the same output class are listed together followed by another set of rules of the same output class. This method of ordering the rules has two advantages: (1) each rule in the output sequence can be interpreted independently from the other rules of the same class and (2) when referring to a rule in the sequence, instead of considering that none of the situations of the above rules has evaluated to true, it would be enough to consider that none of the classes above the current class has been the output class. Good improvement can be achieved by sorting the rules by their classes, while additionally speeding up the simplification algorithm.

As the simplification algorithm has been presented to simplify TR sequences, it assumes that the attributes are Binary. Therefore the next issue in applying the simplification algorithm to classification rules is how to convert the attributes with more than two nominal values into binary attributes. Among the possible ways of doing such a conversion are what we call *Binary* and *Unique* encoding. The former uses a binary code to represent nominal values of each attribute. For instance if an attribute *a* has three nominal values of 1, 2, and 3, then they can be represented, respectively, by 00, 01, and 1x, which means that the non-binary attribute *a*, can be replaced with two binary attributes. In unique encoding, however, more binary attributes are used, one per nominal value, e.g. 100 for 1, 010, for 2, and 001 for 3. In general, in unique encoding, each attribute *a* with *k* nominal values of $v_1, v_2, \ldots, v_k$ is represented with *k* binary attributes $a_1, a_2, \ldots, a_k$, such that the value of a_j is 1 if, and only if, the value of *a* is v_j.

Although the Binary encoding method uses fewer bits than the unique encoding method, a potential disadvantage with it could be that it allows more than one rule to be represented as a single rule. For instance if an attribute *a* can be either 1, 2, or 3, represented respectively by 00, 01, and 1x, then value 0x for *a* would mean 'if *a*=1 or *a*=2', which is in fact the combination of two rules (recall that each situation is a conjunction-not disjunction-of literals). Values like 0x may occur as a result of simplification or the application of a rule induction algorithm. So we prefer the unique encoding method over the Binary encoding method in this report.

A possible approach to simplifying a given set of classification rule is to treat each rule as a data sample and perform the following steps:

(1) encode the rules
(2) run a rule induction algorithm, such as ID3 to extract the rules
(3) run the simplification algorithm
(4) decode the rules (optional)

If the rules are decoded in the last step, then they will be based on the original non-binary attributes; otherwise they will still be in terms of the binary attributes.

However, running a rule induction algorithm directly over the rules usually results in more rules, e.g. a larger ID3 tree, compared to the case where the original samples are used, because each rule represents several samples, hence less "freedom" for the rule induction algorithm in general. Nevertheless, such an approach could be advantageous when the given set of rules has a desirable accuracy which one does not want to loose.

Another possible approach is simply the application of the simplification algorithm together with encoding and decoding steps:

(1) encode the rules
(2) run the simplification algorithm
(3) decode the rules (optional)

The unique encoding method would not be suitable for this approach, even though it was so in the previous approach, because it results in too many don't cares. Having don't cares was not a problem in the previous approach as a rule induction algorithm, such as ID3, makes use of them to produce a better output, e.g. a smaller ID3 tree. However, it is a problem in this approach, because the simplification algorithm does not use them as it is not supposed to affect the coverage of the samples. Therefore we introduce another type of encoding, called *x encoding*, which is the same as unique encoding except that it uses 'x' (for don't care) instead of '0'.

Example 11. Suppose that an attribute has three nominal values of 1, 2, and 3. Using unique encoding we will have 100 for 1, 010 for 2, and 001 for 3. So we will have 2^3-3=5 don't cares, which are: 000, 011, 101, 110, and 111. However, using *x*-encoding, we will have 1xx for 1, x1x for 2, and xx1 for 3, and the only don't care of 000.

In general, if an attribute has *k* nominal values, the number of don't cares in the unique encoding method will be 2^k-*k*, whereas it would only be 1 in the x-encoding method. A feature of x-encoding is that each attribute is used at most once in each rule. For instance, if an attribute *a* can be 1, 2, or 3,

there will be at most a single *a*-test of '*a*=1', '*a*=2', or '*a*=3' in each rule. However, using unique encoding a rule may contain tests like '$a \neq 1$ *and* $a \neq 3$' in the same rule.

In order to make use of the remaining don't cares in the x-encoding method, we add some dummy rules at the top of the given rules before applying the simplification and remove them afterward. This allows the simplification algorithm to take the advantage of the don't cares in order to achieve a more simplified output. Let us assume that the don't care resulted by applying the x-encoding method to an attribute *a* with *k* nominal values corresponds to a special "virtual" value denoted by x_a. Clearly, the attribute *a* can never take the value x_a. For instance, x_a for attribute *a* in Example 11 will be represented as 000. Then, we use the following steps:

(1) Add the dummy rules at the top:
    For each attribute *a*, do:
        Add ($a=x_a$)→"dummy" at the top
(2) Apply the x-encoding method
(3) Run the simplification algorithm
(4) Remove the dummy rules:
    For each rule *r*, whose action is "dummy" do
        Remove *r*
(5) Decode the rules into the original format

8. Experimental Results

To test the simplification algorithm, we used Car Evaluation and Monk's first datasets [8]. The datasets and their description can be found at the UCI machine learning repository (http://www.ics.uci.edu/~mlearn/MLRepository.html). In order to obtain TR sequences to simplify, we first encoded the datasets and ran (Binary) ID3. We used the resulting sets of rules as TR sequences to simplify. Since the simplification algorithm is order-dependent, we ran it three times each time with a different ordering of the input rules. Table 4 shows the results. Note that the simplification algorithm does not change the accuracy of the given set of rules.

Table 4.

Problem	#rules #attributes in samples	#rules(#tests) before simplification- accuracy%	#rules(#tests) after simplification run 1	#rules(#tests) after simplification run 2	#rules(#tests) after simplification run 3	Average of #rules(#tests) and their reductions%
Car Evaluation (Unique)	1728 6	79(776) 100%	47(304)	54(321)	60(371)	53.67(332) 32.1% (57.2%)
Car Evaluation problem (Binary)	1728 6	127(1105) 100%	35(104)	50(162)	48(148)	44.3(138) 65.1% (87.5%)
Monk's first (Unique)	124 6	21(142) 92.59%	15(57)	17(40)	12(40)	14.67(45.67) 30.2% (67.8%)
Monk's first (Binary)	124 6	34(203) 87.73%	31(80)	27(79)	25(82)	27.67(80.33) 18.6% (60.4%)

Table 4 shows that the simplification algorithm could remarkably reduce the size of its input. We noticed that smaller outputs usually (but not always) correspond to the cases where the input TR sequence is complete and there are relatively a great number of rules with the same class (i.e. the same action) at the bottom of the input TR sequence. This is because all such rules will be replaced with a single rule of 'T→the class' by the simplification algorithm. However, it might be fairer to compare the output of simplification with a version of the input set of rules which includes the *default to major rule* (T→major class as the last rule) [9]. Table 5 provides this comparison for both Car Evaluation and Monk's first problems.

Table 5.

Problem	Before simplification	Using T→ Major before Simp.	Using T→ Major after Simp.
Car Evaluation (Unique)	79(776) 100%	44(413)	36(225)
Car Evaluation problem (Binary)	127(1105) 100%	68(572)	34(173)
Monk's first (Unique)	21(142) 92.59%	10(62)	10(50)
Monk's first (Binary)	34(203) 87.73%	16(92)	16(75)

In the next series of experiments, we used the *inducer* software [9] to receive the classification rules for Monk's first problem using both the standard ID3 and Prism algorithms [10]. Then we applied the x-encoding method followed by the simplification algorithm to simplify the induced rules. We performed the experiment for both cases of with and without the default to major rule. Table 6 shows the number of rules and literals in different cases.

Table 6.

Problem	Before simplification	After Simp. Without using Dummy rules	After Simp. With using Dummy rules
Monk's first- not complete (ID3)	52(226) 76.6% correct 10.4% not covered	50(199)	37(130)
Monk's first-complete using T→Major class (ID3)	27(110) 85.9% 0% not covered	22(89)	22(85)
Monk's first- not complete (Prism)	25(75) 87% correct 13% not covered	21(60)	15(35)
Monk's first-complete using T→Major class (Prism)	6(10) 100% 0% not covered	5(7)	5(7)

Table 6 suggests that the simplification algorithm might be beneficial in reducing the number of classification rules or at least the number of the tests in a set of classification rules, but further experiments is needed in this regard.

Comparing Table 4 and Table 6, on the Monk's problem, another interesting result can be inferred, which is independent from the simplification algorithm. It can be seen that using the unique encoding method together with (Binary) ID3 has the following advantages over using just the standard ID3:

- Higher percentage
- Providing complete rules set, i.e. no missing link in the ID3 tree
- Smaller size, e.g. fewer rules and literals

These advantages could be due to the fact that the unique encoding method provides higher expressive potential. Because of these advantages, the following could be a useful replacement for the standard ID3:

(1) Apply the unique encoding method to the samples
(2) Apply (Binary) ID3 to the encoded samples and derive the rules
(3) Decode the rules

However, further research is needed to evaluate this replacement. A possible difficulty with this approach might be when the underlying dataset contains attributes with too many nominal values.

9. Conclusion

In this report, we presented two algorithms to remove redundant rules and literals from a given TR sequence. Then we draw the main simplification algorithm by combining the algorithms. The simplification algorithms can also be applied to decision lists, as a decision list is a special type of a TR sequence. The algorithms may also be used in order to reduce the size of a given set of classification rules by converting it to a sequence of (ordered) rules. Some methods for this purpose were also proposed in this report. Although the algorithms remove redundant rules and literals from a given TR sequence, they do not try to achieve a more simplified output through re ordering the rules.

References

[1] N.J. Nilsson, "Teleo-Reactive Programs for Agent Control," *Journal of Artificial Intelligence Research*, 1, 1994, pp. 139-158.

[2] N.J. Nilsson, *Learning Strategies for Mid-Level Robot Control: Some Preliminary Considerations and Results,* Robotics Laboratory, Department of Computer Science, Stanford University, May 2000

[3] M. Sahami, "Learning Classification Rules Using Lattices," *Proceedings of the Eighth European Conference on Machine Learning (ECML-95),* Springer-Verlag, Berlin, Germany, 1995, pp. 343-346.

[4] M.A. Bramer, "Using J-Pruning to Reduce Overfitting in Classification Trees," *Research and Development in Intelligent Systems XVIII* Springer-Verlag, 2000, pp. 25-38.

[5] M.A. Bramer, (ed.), *Special Issue of Knowledge Based Systems Journal,* vol. 15, Issues 5-6, Elsevier, 2002.

[6] L.R., Rivest, "Learning Decision lists," *Machine Learning* 2,3, 1987, pp. 229-246.

[7] J.R. Quinlan, "Induction of Decision Trees," *Machine Learning,* 1, 1986, pp. 81-106.

[8] S.B. Thrun et al., *The MONK's Problems-A Performance Comparison of Different Learning Algorithms,* tech. report CMU-CS-91-197, Carnegie Mellon University, 1991.

[9] M.A. Bramer, "Inducer: a Rule Induction Workbench for Data Mining," *Proceedings of the 16th IFIP World Computer Congress Conference on Intelligent Information Processing (eds. Z.Shi, B.Faltings and M.Musen),* Publishing House of Electronics Industry (Beijing), 2000, pp. 499-506.

[10] M.A.Bramer, "Automatic Induction of Classification Rules from Examples Using N-Prism," *Research and Development in Intelligent Systems XVI*. Springer-Verlag, 2000, pp. 99-121.