Learning with C4.5 in a Situation Calculus Domain

Kamran Karimi and Howard J. Hamilton
Department of Computer Science
University of Regina
Regina, Saskatchewan
Canada S4S 0A2
{karimi,hamilton}@cs.uregina.ca

Abstract: It is desirable to automatically learn the effects of actions in an unknown environment. Using situation calculus in a causal domain is a very natural way of recording the actions and their effects. These could later be used for Automatic Programming purposes. A brute force approach to representing the situations involves recording the value of all the available variables. This is a combinatorial problem, and becomes unmanageable when the number of variables, or their domains, gets too large. A solution is to represent the situations and the transitions among the situations using first order logic formulas, which allows for generalizations and removal of irrelevant variables. But this usually requires a domain expert to act as a programmer. In this paper we look at the problem of automatic learning of causal and association rules in a situation calculus domain from observations, with little help from a domain expert. We feed C4.5 with temporal data generated in an Artificial Life environment, where the results of taking an action are not known to the creatures living in it. We show that C4.5 can discover causal rules, and has a good ability to prune irrelevant variables. We have modified C4.5 to generate Prolog rules as output, and show how the automatically learned rules can be used to make plan generators. The presented approach succeeds in generating plans in deterministic environments and needs less domain dependent preprocessing than techniques like Reinforcement Learning or Genetic Programming.

1. Introduction

Situation Calculus [6] is a method of describing the effects of actions. Each situation can be considered a snapshot of the values of a set of variables. One can move from a situation to another by performing actions. This can result in a possibly cyclic graph with situations as nodes and actions as links between the nodes. One can interpret the transitions between the situations as the execution of rules. The starting situation forms the precondition of the rule, and the resulting

situation forms the results. Planning is easy here: To go from the current situation to a desired situation, first make sure that they are both in the graph, and then find a path connecting them. Following this path can be regarded as executing the plan [8]. One problem with this brute-force approach to representing the effects of actions is that the number of possible situations grows exponentially as the number of variables increases or their domains become larger. This can make representing a graph very expensive or even impossible.

At the other end of the spectrum one can represent the situations and the transitions among them using first order logic formulas. [7] suggests that logic is a an appropriate way of representing knowledge. Suppose do(A, S) means performing action A in situation S, with the result being another situation. A statement like has(O, do(pick(O), S) would then mean that if in any situation the agent picks up an object O, then it has the object O in the next situation.

This method of representation allows us to generalise across variable values because the statement is true for many values of O and S. It also generalises across variables themselves, because the statement holds irrespective of what other variables (other than O and S) hold at the time. This form of representation has been used for programming purposes. For example, in [5] rules extracted in a situation calculus domain are considered as logic programmes. In GOLOG [3] which is a programming language based on Situation Calculus, the programmer writes code to specify the initial state of the environment, the preconditions and the effects of actions. This approach results in efficient representations of the domain. Our aim is to simplify the process of writing plans. We attempt to do this by deriving the rules of the environment automatically and then using them as parts of a plan generator.

Here the goal is to discover rules in a simplified world by using observational data and without programming. We will use C4.5 [9] for this purpose because it is widely available and produces the kind of rules that is suitable for a situation calculus domain. A situation transition can readily be represented by a simple **if-then** rule such as **if**{(<current situation> AND <action>)} **then** (<next situation>). We will see that if we give C4.5 enough information in the form of the relevant variables, it might be able to produce usable rules automatically. Though this may not result in very efficient representation in terms of the number of rules generated, it does remove the need for actual programming. The steady increase in computing power justifies the willingness to accept some redundancy in exchange for less work by the user.

The paper is divided into two main parts. First we show that C4.5 is suitable for discovering the effects of actions in a Situation Calculus domain. However, as individual rules, C4.5's results are of limited use, so we go on to show how Prolog can be used to turn C4.5's output into executable parts of a plan. The emphasis will be on reducing human involvement as much as possible.

The rest of the paper is outlined as follows. Section 2 describes our target domain, which is an Artificial Life [4] simulator. Section 3 shows how C4.5 performs when

fed with temporal data generated in this simulator. The results show a strong generalization ability. Section 4 describes how C4.5's output in Prolog can be used to build plan generators. This turns C4.5's output into executable code. We will see how the temporal characteristics of the generated rules allow Prolog to perform recursive searches. The approach presented in this paper is not perfect, so in Section 5 we summarize some of its strong and weak points.

2. An Artificial Creature's Learning Problem

A state S_i is defined as a tuple of attributes $<x_{i1}, x_{i2},\ldots x_{ij}>$. We go from one situation to another by performing actions over time. This means that we have a temporally ordered sequence of situation transitions caused by actions. We include the action as part of the situations in the sequence, and get records of the form $<x_{i1}, x_{i2},\ldots x_{ij}, a_i>$. The aim is to know the effects of actions in different situations. We consider the *time window t* to be the number of situations that may be causally related. With a time window of 3, for example, the effects of an action taken in a situation will be known in the next 2 situations.

We use an artificial environment called URAL [11] as our source of data. The world in URAL is made of a two dimensional *n* by *m* board with one or more agents living in it. These are called *creatures* in Artificial Life literature. A creature moves around and if it finds food, eats it. There can be obstacles on the board. Food is produced by the simulator and placed at positions that are randomly determined at the start of each run. The creature can sense its position, humidity and temperature, and also the presence of food at that position. Humidity and temperature are constants that are assigned by the simulator to each location on the board. At each time-step the creature randomly chooses to move from its current position to either Up, Down, Left, or Right. The creature does not know the meaning or results of any of these actions, so for example it does not know that it's position may change if it chooses to go to Left (action L). It can not get out of the board, or go through the obstacles that are placed on the board by the simulator. In such cases, a move action will not change the creature's position. The creature can sense which action it takes in each situation. The goal is to learn the effects of the actions. This could later be used for making plans in the form of a series of movements from the set {U, D, L, R}, to reach food. URAL's creatures build situation calculus graphs and use them to store their observations of the world and to make plans for finding food.

For our creature time passes in discrete steps. At each time step it takes a snapshot of its sensors and randomly decides which action it should perform. This results in records such as *<x position, y position, is food here?, temperature, humidity, action>* which the creature can log to a file. Listing 1(a) shows an example sequence of such records, with time passing vertically from top to bottom. These records can be used to find association among variable values. One example is the association between the presence of food and a specific location. Discovering causal rules, however, needs considering the passage of time, as in general the effects of an action such as moving appear at a later time. URAL can output more

than one consecutive record as a single record, and set the last variable to any attribute. We call this operation *flattening* the records. C4.5 does not understand time, and this pre-processing step enables it to work on a window of two or more records instead of a single one. Thus time will be implicitly included in the records. Flattening the sequence in Listing 1(a) using a time window of size 2 and with x as the last variable would give us the records in listing 1(b). The subscripts show the time-step from which the variable was taken. We treat the last attribute specially because C4.5 considers it to be the decision attribute.

<x, y, f, t, h, a >	<$x_1, y_1, f_1, t_1, h_1, a_1, x_2$ >
<1, 3, false, 43, 24, L>	<1, 3, false, 43, 24, L, 0 >
<0, 3, false, 26, 32, L>	<0, 3, false, 26, 32, L, 0 >
<0, 3, true, 26, 32, D,>	<0, 3, true, 26, 32, D, 0>
<0, 4, false, 12, 65, U>	
a	b

Listing 1. (a) A sequence of two records. (b) The flattened sequences.

In a sequence of flattened records, time passes horizontally from left to right as well as vertically from top to bottom. We have chosen a time window of 2 because we know that the effects of a move action will be known in the next time step. Here we have renamed the variables to remove name clashes. Flattening can be done for any attribute value of interest. Listing 2 shows the first two records in Listing 1(a) flattened with y, f, t, and h as the decision attribute.

<$x_1, y_1, f_1, t_1, h_1, a_1, y_2$>: <1, 3, false, 43, 24, L, 3>
<$x_1, y_1, f_1, t_1, h_1, a_1, f_2$>: <1, 3, false, 43, 24, L, false>
<$x_1, y_1, f_1, t_1, h_1, a_1, t_2$>: <1, 3, false, 43, 24, L, 26 >
<$x_1, y_1, f_1, t_1, h_1, a_1, h_2$>: <1, 3, false, 43, 24, L, 32 >

Listing 2. Flattening with a time window of 2 for different decision attributes.

3. Rule Generation with C4.5

A creature needs to move to specific places. It cannot directly set the values of x and y to a desired value, but can perform actions through its actuators that change the values of x and y. What it needs is a set of rules to tell it when to apply a certain action to get closer to its goal. We are interested in finding rules for the next values of x and y, and also for the place of food. The correct condition attributes for x_2 are x_1 and a_1, and for y_2 they are y_1 and a_1. For f_2 any set of attributes from the second time step is correct, but the desired attributes are x_2 and y_2 because they are under indirect control. Humidity and temperature are constants for each location.

This means that t and h have many of the same characteristics as x and y and could creep up into the rules, which is undesirable. In [1] we used C4.5 to generate correct rules for food and the next values of x and y. In that paper the creature was allowed to completely explore the world before its observations were fed to C4.5.

3.1 No Obstacles and a Time Window of 2

Here we investigate the effects of incomplete coverage of the world. The results for one typical run in a 15 by 15 board with no obstacles are shown in Table 1. "Correct" means having only a_1 and x_1 (for x_2) or a_1 and y_1 (for y_2) as condition attributes.

Moves	Nodes	Links	Decision	Rules	Correct
500	116	263	x_2	52	51
			y_2	47	47
1000	185	472	x_2	59	59
			y_2	60	60
1500	199	573	x_2	60	60
			y_2	60	60

Table 1. C4.5's results for a world with no obstacles.

Visiting more nodes and links increases the number of rules until they settle down at 60. For comparison, a brute force approach that only considered the x and y variables would have 225 nodes and a maximum of 900 (225×4) links. Each link would be equivalent to one rule. The rules for moving along the x and y axis are independent of each other, so the maximum number of distinct rules for each of x and y is 60 (15×4) which is the same as what we got. C4.5 did not need to visit all of the world to come up with rules that were correct everywhere because the same rules hold in all columns and rows. With smaller number of moves, most of the wrong rules looked like this: **if** $\{(x_1 = 9)\}$ **then** $(x_2 = 9)$, which suggests that C4.5 could not figure out how to move around in a certain column or row of the world because it was not explored enough.

We derived rules for the places of food too. Some of the rules discovered for f_2 contained t_2 and h_2 in addition to x_2 and y_2. There is no causality relation here, and these results are perfectly right from an association point of view. The problem is, there is no way for the creature to "affect" the temperature or humidity, and get to the food. This is what we call the *Association Problem*: C4.5, not knowing about the semantics of the domain, uses variables with strong association with the decision attribute in the rules it generates. C4.5's goal is not to come up with "meaningful" rules, but to use all available variables to come up with the shortest possible decision tree. Knowing that it is possible to change x and y only, the user can for example exclude temperature and humidity when outputting the flattened

record for food. Since flattening is not necessary when looking for associations, URAL can be instructed to output records of the form <x, y, a, f>.

3.2 Obstacles and a Time Window of 2

We tried C4.5 when there were 10 randomly placed obstacles on the board. This resulted in an increase in the number of generated rules, as C4.5 now had to make rules that considered the exceptions to the general laws of moving around the board. The association problem was seen here too. C4.5 attempted to identify the position of the obstacles not only by their x and y values, but also by the humidity and temperature. We genereted new test data and allowed C4.5 to only choose from x_1, y_1 and a_1 variables. We expected an increase in the number of rules because C4.5 now had fewer variables to work with. However, there were not any significant increase in the number of rules. Results of two typical runs, one with all the variables, and one with only the position and action variables, come in Table 2(a) and 2(b), respectively.

Moves	Nodes	Links	Decision Attribute	Rules
500	101	231	x_2	63
			y_2	35
1000	165	405	x_2	64
			y_2	62
1500	175	460	x_2	64
			y_2	60
4000	215	731	x_2	79
			y_2	76
6000	215	763	x_2	80
			y_2	75
7000	215	763	x_2	83
			y_2	75

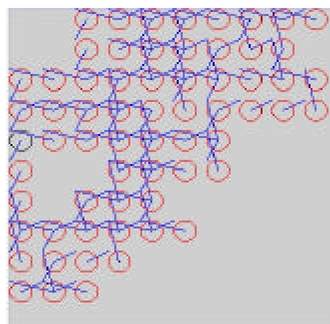
a

Move s	Nodes	Link s	Decision Attribute	Rules
500	123	253	x_2	56
			y_2	62
1000	153	395	x_2	58
			y_2	66
1500	190	520	x_2	64
			y_2	70
4000	215	740	x_2	84
			y_2	78
6000	215	761	x_2	84
			y_2	82
7000	215	761	x_2	84
			y_2	82

b

Table 2. C4.5's typical results with obstacles on the board.

The number and correctness of the results depend heavily on not only the number of nodes visited, but also on the number of links travered. The number of links tell us how much the creature knows about ways to move between the situations. The more connections the creature is aware of, the better. In URAL, we can see which x and y values have been explored by looking at the creature's "brain" in a window as shown in snapshot 1. The move direction is usually determined randomly, but the user can also guide the creature to specific locations by "manual control" via on-screen buttons that instruct it to move in a specific direction. This is like allowing a robot to experience randomly with its actuators, but also having a joystick to guide it into performing certain interesting actions that it may have missed to perform on its own.



Snapshot 1. The creature's brain shows which locations have been explored.

Each circle in snapshot 1 denotes one (x, y) position in the world. The empty locations have not been explored. The lines show the paths taken by the creature as it has moved from one location to the next (the lines start at the centre of the source location). The denser this graph, the better the resulting rules. For the creature to find correct and general rules, it should have entered the same spot from all existing neighbours. That is when C4.5 will have enough knowledge to generate rules that are general enough to always work. In the case of an obstacle or at the borders of the world, the creature should have tried moving to the place of the obstacle, and experienced the failure. In these cases, C4.5 would create rules like **if** {($x_1 = 9$) AND ($a_1 = \mathrm{R}$)} **then** ($x_2 = 9$). This is an exception to the general rule that moving to the right increases the value of x.

3.3 No Obstacles and Time Windows Greater Than 2

In order to see how sensitive the results are to the selected time window, we tried different time window values from 2 to 10 with no obstacles and all the condition variables present. The world was completely explored. The results are in Table 3. Only 3 cases are displayed because the results followed the same pattern for all other values of the time window. The subscripts of the variables designate the time step during which they were observed.

Window	Total Attributes	Decision Attribute	Condition Attributes
2	6	x_2	a_1, x_1
	6	y_2	a_1, y_1
6	30	x_6	a_5, x_5
	30	y_6	a_5, y_5
10	54	x_{10}	a_9, x_9
	54	y_{10}	a_9, y_9

Table 3. Condition attributes selected by C4.5 for different time windows.

With a time window of t, C4.5 consistently chose attributes from time step t-1 as the determining factors for the decision attribute. This shows that in the presence of a strong causal relation, the results are not sensitive to the time window.

3.4 Obstacles and Time Windows Greater Than 2

We evaluated the results after 7000 moves for time windows of 2 to 10. As in section 3.3, adding obstacles to the board did not affect C4.5's ability to choose the condition variables from time step t-1. The number of rules generated when C4.5 had access to all the previous variables varied slightly for different time steps, but they were the same when the available condition attributes were limited to x, y and a.

4. Using Prolog for Plan Generation

Here we consider a plan and a programme to be the same. In the previous section we were concerned about learning the immediate effects of individual actions. Now that we know these effects, we can combine the actions in the form of a plan and come up with more complex behaviours. Most other work on planning starts at this point, because they consider the problem of knowing the effects of actions as already solved. We have modified C4.5 to output the discovered rules in Prolog [10] statements that can be directly fed to a Prolog interpreter. A new option '-p' has been added to the c4.5rules programme that causes it to create a <file stem>.pl Prolog file in addition to its normal output. The generated Prolog statements are in the Edinburgh dialect.

There are two main problems in using Prolog to represent C4.5's rules. First, C4.5 assigns certainty values to the rules it outputs. These can not be represented in standard Prolog, and so are ignored. In a domain with little "surprises," the certainty value is very high and we expect this problem to be of minor consequence. This condition holds in URAL and many other industrial applications of robots. The second problem is that there is no concept of time in standard Prolog. But as explained later, using Prolog to represent the rules is especially appropriate in temporal domains, where the decision attribute is actually one of the condition attributes, seen at a later time. A temporal order is implicitly present in Prolog because it follows a rule's conditions from left to right. Table 4 shows parts of an example set of statements generated by the modified c4.5rules programme when the decision attribute is x_2.

class(A1, X1, 0) :- A1 = 2, X1 = 1.
class(A1, X1, 2) :- A1 = 3, X1 = 1.
class(A1, X1, 3) :- A1 = 2, X1 = 4.

Table 4. Three sample Prolog statements generated by c4.5rules.

In Table 4 a value of 2 and 3 for action *A1* could mean going to the left and right, respectively. Following C4.5's terminology, the results are designated by a predicate called "class." The condition attributes (action *A1* and position *X1* in this case) come first, and the value of the decision attribute (the next value of x) comes last. In the head of the rules, the condition attributes are used for the decision making process. In our example temporal data, *A1* and *X1* belong to the current time step, while the classification is done for the value of x in the next time step. To use such rules the user can issue queries like class(2, 4, X2) (where does the creature go from $x = 4$ if it moves Left?). If we are dealing with more than one sensor variable (x and y for example) we could rename "class" to something like "classx" to avoid name clashes.

Notice that the automatically generated Prolog statements use the unification operator (=) instead of the comparison operator (=:=). This allows the user to traverse the rules backward and go from the decision attribute to the condition attributes, or from a set of decision and condition attributes, to the remaining condition attributes. Some example queries are class(A1, 1, 2) (which actions take the creature from $x = 1$ to $x = 2$?) or class (A1, X1, 3) (which action/location pairs immediately leads to $x = 3$?). This makes C4.5's discovered rules generally more useful.

C4.5 can generate rules that rely on threshold testing and set membership testing. If we use the standard Prolog operators of =< and > for threshold testing, and implement a simple member() function for testing set membership, then we would not be able to traverse the rules backward, as they lack the ability to unify variables. So if we had a clause like: class(A, B, C) :- A =< B, member(A, C), then we would be unable to use a query like class(A, 3, [1, 2, 3]), because Prolog can not perform the test =< on variables that are not unified. Adding the unification ability to =<, > and member() will remove this limitation. For example, *X* > 10 would choose a value above 10 for *X* if it is not already unified, and member(*X*, [1, 2, 3]) would unify *X* with one of 1, 2, or 3 if it is not already unified. Both cases would always succeed if *X* is not unified, but could fail if it is. We have written some simple code to do just this and the results are shown in Table 5. We employed a deterministic method to choose the value of the variable that is going to be unified, but one could use a random method too. ule (unify less-equal), ug (unify greater) and umember() (unify member) are the unifying counter parts of =<, > and member(), respectively.

<table>
<tr><th>Name</th><th>Unifies?</th><th>Implementation</th></tr>
<tr><td>=<</td><td>No</td><td>Standard</td></tr>
<tr><td>></td><td>No</td><td>Standard</td></tr>
<tr><td>member()</td><td>No</td><td>member(A, [A|_]).
member(A, [_|B]) :- member(A, B).</td></tr>
<tr><td>ule</td><td>Yes</td><td>:- op(800, xfx, ule).
A ule B :- var(A), A = B.
A ule B :- A =< B.</td></tr>
<tr><td>ug</td><td>Yes</td><td>:- op(800, xfx, ug).
A ug B :- var(A), A is B + 1.
A ug B :- A > B.</td></tr>
<tr><td>umember()</td><td>Yes</td><td>umember(A, B) :- var(A), [X|_] = B, A = X.
umember(A, B) :- member(A, B).</td></tr>
</table>

Table 5. Prolog operators and functions for planning.

If the argument to the left of ule is not unified, ule sets its value to be the same as its argument to the right, and returns with success. If the left hand side argument is already unified, then it does a =< test. The ug operator does the same with regard to >. If unification is needed, it sets the left hand side argument to the value of the

right hand side argument plus 1. In both cases the right hand side argument should already have been unified. The function umember() unifies the first argument with the first member of the list if unification is needed. The second argument to this function should have already been unified. These conditions seem to hold for rules generated by C4.5. The modified c4.5rules programme can thus generate rules of the form class(A, B, C, 0) :- A = 1 , B ug 10, umember(C, [1, 2, 3]). As explained below, this unification ability has another advantage: it allows us to generate plans by moving backwards in the rules.

To create a Prolog plan generator, we have to make manual modifications in the Prolog statements generated by c4.5rules. The results will be a set of rules that search backward from a desired situation to the current situation, and if such a path is found, it prints the actions that have to be performed to get to the desired situation. The modifications should be done manually because we have made the changes to c4.5rules in a general manner and compatible with the normal output of C4.5. The Prolog clauses that are generated simply do a normal classification without caring about any "bigger picture" that may exist in a particular application such as planning. We now go over the modifications needed for planning. Suppose we start with the rule: class($A1$, $X1$, 0) :- $A1$ = 2, $X1$ = 1.

1) We have to make sure that Prolog does not get stuck in plans with cycles. To prevent this we keep track of the classes we have already visited. This is done by adding a list variable to the class() clause, and adding code for cycle-prevention. So now we have this rule: class($A1$, $X1$, 0, P_1) :- $A1$ = 2, $X1$ = 1, not(umember($X1$, P_1)), P_2 = [$X1$|P_1].

   $X1$ is used to distinguish among the steps in the plan. If we do not find a value of $X1$ in our list, then we know we are not in a cycle. If this holds we add it to the list P_1 to get a new list P_2, and continue from there.

2) In a temporal domain one of the condition attributes may actually be the previous value of our decision attribute. In this example x has this property. We now have to make this fact explicit, because we want Prolog to make sure that we are actually in the previous situation before advancing to the next one. To do this we introduce the class() clause in the condition part of the Prolog statements. This will allow Prolog to use recursion and try all the paths that leads from one class to the next. So we now have: class($A1$, $X1$, 0, P_1) :- $A1$ = 2, $X1$ = 1, not(umember($X1$, P_1)), P_2 = [$X1$|P_1], class(_, _, 1, P_2).

   Notice that the value 1 in class(_, _, 1, P_2) comes from $X1$ = 1, as in this example class() is C4.5's name for the x position. We would do the same for other sensor variables. Now Prolog can search for a way to get the robot from the starting state to a desired state.

3) We can add a statement to print the plan after it is generated. This is simple to do: class($A1$, $X1$, 0, P_1) :- $A1$= 2, $X1$ = 1, not(umember($X1$, P_1)), P_2 = [$X1$|P_1], class(_, _, 1, P_2), printOut($A1$, $X1$).

4) Finally, we introduce the current situation as a class statement. For example, if we are currently at $X1 = 0$, we add the clause class(_,_,0,_) to the set of rules. Intuitively this means we are currently at position 0, and we do not care how we got here.

Automating the above steps requires c4.5rules to know the previous value of the decision attribute. Conveying this information to c4.5rules is not very easy, so the transformation is done manually. The programme could of course be modified specifically for this problem to automatically generate rules of the form: class(*A1*, *X1*, 0, P_1) :- $A1 = 2$, $X1 = 1$, not(umember(#, P_1)), P_2 = [#|P_1], class(_, _, *, P_2), printOut(*A1*, *X1*) and rely on the user to edit them and replace the '#' and '*' markers with the appropriate attributes.

Performing the above steps on the rules in Table 4 gives us the rules in Table 6. The helper functions not() and printOut() are also provided. The function umember() has already appeared in Table 5.

class(_, _, 0, _).
class(*A1*, *X1*, 0, P_1) :- $A1 = 2$, $X1 = 1$, not(umember(*X1*, P_1)), P_2 = [*X1*|P_1], class(_, _, 1, P_2), printOut(*A1*, *X1*).
class(*A1*, *X1*, 2, P_1) :- $A1 = 3$, $X1 = 1$, not(umember(*X1*, P_1)), P_2 = [*X1*|P_1], class(_, _, 1, P_2), printOut(*A1*, *X1*).
class(*A1*, *X1*, 3, P_1) :- $A1 = 2$, $X1 = 4$, not(umember(*X1*, P_1)), P_2 = [*X1*|P_1], class(_, _, 4, P_2), printOut(*A1*, *X1*).
not(*G*) :- *G*, !, fail. not(*G*).
printOut(*A*, *X*) :- write('Robot is at: '), write(*X*), write(', it does action: '), write(*A*), nl.

Table 6. Prolog rules modified for planning.

Using the above statements, the user can perform Prolog queries of the form class(_, _, 7, []) to find a plan. He can include a position in the last argument of his query to prevent that position from showing up in the plan, as anything in that list will be avoided. Prolog will then print out the actions that should be performed to get from the current situation $x = 0$ to the desired situation $x = 7$.

The association rules for the presence or absence of food can use any condition variables, including the ones that do not make sense. One example is given in Table 7 below.

class(X, Y, 1) :- X = 6, Y = 7.
class(X, Y, 1) :- X = 2, Y = 7.
class(X, 0) :- X = 0.
class(Y, 0) :- Y = 0.
class(Y, 0) :- Y = 1.
class(A, 0) :- A = 1.

Table 7. Prolog rules for the presence of food.

Since we only care about the presence of food, the rules that will actually be executed are in the first two rows, with a 1 as the classification value. This means that the fact that the last rule does not make sense (it claims we can *not* find food if we perform an action) is not important. Such useless association rules are expected to be present, since in URAL it is the presence of food that follows a pattern of appearing at certain (x, y) positions, while this does not apply to the absence of food.

The first rule, when modified to form a plan, will look like: classf(1) :- classx(_, _, 6, []), classy(_, _, 7, []). This is an association rule, so there is no recursion here. Note that we have renamed the class() names to avoid clashes. Invoking the classf(1) command in Prolog will cause the planner to come up with a sequence of moves to get the creature to a place of food.

5. Concluding Remarks

For us ease of use is very important. With our approach very little pre-processing was needed to produce the rules that expressed the effects of actions in different situations. All we did was to feed the observed records directly to C4.5. All the pre-processing that was done involved choosing the decision attribute, and pruning unrelated condition attributes, so C4.5 would be able to come up with useful rules. In URAL this was done by clicking a few checkboxes before saving the observed records to a file, so there was no need to involve a programmer in the process. This is unlike some other Automatic Programming techniques like Reinforcement Learning or Genetic Programming where a domain expert must explicitly provide the system with high-level information. For Reinforcement Learning the work includes deciding on the behaviours that should be reinforced, and a payoff function, among others. For Genetic Programming one should determine sets of terminals and primitive functions and come up with a suitable fitness function. A discussion of this aspect of Genetic Programming and Reinforcement Learning appears in [2]. Implementing a system to actually produce the results comes next. There is no guarantee that performing these pre-processing steps will be easier than writing a program manually.

Our approach requires little explicit programming, which sets it apart from programming systems like GOLOG, which require the user to explicitly encode all the knowledge of the domain into Situation Calculus rules. The main problem with our approach is that a set of simple rules as generated by C4.5 may not have the ability to express the rules of a complex environment. This means that such rules are more suitable for expressing basic operations such as a single move action. As is evident in the case of the Prolog planner in this paper, going beyond the basic operators to write a programme that can guide a creature needs some manual work. Another problem is that for this approach to work, all situations of interest should have been experienced. This could be very hard or impossible to do as the search

space becomes bigger. A programming approach like GOLOG is probably more suitable in very complex environments.

In the presented approach the domain expert can decide to write plans manually if it is necessary to do so. For example, doing a certain experiment may have undesirable consequences for the creature or the environment. In such cases the domain expert can opt to pre-programme the creature to avoid the problem, and let it discover other rules.

One thing to consider is that C4.5 can not generalise across variable values, so we will not have rules that literally look like: **if** $\{(x = \mathbf{a})$ AND $(a = \mathrm{R})\}$ **then** $(x = \mathbf{a} + 1)$. This is not a bad thing when there are many exceptions to the rule, because then the generalised rules will have to deal with many special cases. C4.5 allows us to come up with rules that can handle the exceptions, otherwise a human programmer would have to write new rules or modify the existing rules for each randomly generated world because each will have a different set of exceptions. In an approach such as GOLOG, that would amount to manually rewriting a programme for each new environment in which a robot may have to function. It is more interesting to manufacture a robot once and then let it experience its new environment for a while. The observations would then be used to generate plans for the robot.

The modified c4.5rules programme retains backward compatibility, and its output is unchanged when the new option is not used. The modifications are available in the form of a patch file that can be applied to standard C4.5 Release 8 source files. It is available from http://www.cs.uregina.ca/~karimi or by contacting the authors.

Acknowledgements

We thank the anonymous referees for their useful comments.

References

[1] Karimi, K. and Hamilton, H. J. (2000). Finding Temporal Relations: Causal Bayesian Networks vs. C4.5. The 12th International Symposium on Methodologies for Intelligent Systems (ISMIS'2000), Charlotte, NC, USA.

[2] Koza, J. R. and Rice, J. P. (1992). Automatic Programming of Robots using Genetic Programming. The Tenth National Conference on Artificial Intelligence, Menlo Park, CA, USA.

[3] Levesque, H. J., Reiter, R., Lespérance, Y., Lin, F. and Scherl. R. (1997). GOLOG: A Logic Programming Language for Dynamic Domains. Journal of Logic Programming, 31, pp. 59-84.

[4] Levy, S. (1992). Artificial Life: A Quest for a New Creation. Pantheon Books.

[5] Lin F. and Reiter, R. (1997). Rules as actions: A Situation Calculus Semantics for Logic Programs. Journal of Logic Programming Special Issue on Reasoning about Action and Change, 31(1-3), pp.299-330.

[6] McCarthy, J. and Hayes, P. C. (1969). Some Philosophical Problems from the Standpoint of Artificial Intelligence. Machine Intelligence 4.

[7] Moore, R. C. (1985). The Role of Logic in Knowledge Representation and Commonsense Reasoning. Readings in Knowledge Representation, Morgan Kaufmann, pp. 335-341.

[8] Poole, D. (1998). Decision Theory, the Situation Calculus, and Conditional Plans. Linköping Electronic Articles in Computer and Information Science, Vol. 3 (1998): nr 3, http://www.ep.liu.se/ea/cis/1998/008.

[9] Quinlan, J. R. (1993). C4.5: Programs for Machine Learning. Morgan Kaufmann.

[10] Van Le, T. (1993). Techniques of Prolog Programming. John Wiley & Sons.

[11] http://www.cs.uregina.ca/~karimi/URAL.java