Università degli Studi di Bari
Dipartimento di Informatica

LACAM
Machine Learning

CLIPS: meta-rules, decision trees, backward chaining

Antonio Vergari

May 26, 2016

Meta-rules I

As we have seen with searching, designing the rule set for an expert system equals to determine the possible transitions in the space of the WM states.
In more complex context one may want to **create**, **edit** and **delete** rules *at runtime*.
CLIPS provides a primitive for deleting a rule, **undefrule** and one for creating them[1], **build**.

```
1  CLIPS> (build "(defrule foo (a) => (assert (b)))")
2  TRUE
3  CLIPS> (rules)
4  foo
```

However editing them at runtime is not possible in general. If a rule represents a graph search node transition function, one cannot specify a dynamic salience to represent the evaluated value for an heuristic considering the two states at runtime. In the same way as one cannot specify a dynamic and mutable LHS, since it won't compile in a valid part of the RETE, or as one cannot specify an uncertainty factor or define a partial match for the LHS.

---

[1] With it it is possible to evaluate one **def** construct. It is similar to **eval**, while the latter is able to evaluate expressions legal in the REPL. Please refer to CLIPS Basic Programming Guide.

Meta-rules II

To freely manipulate such transitions, that is to be able to govern a more complex strategy, one could adopt a different conceptualization: represent *them as facts*.
In a fashion we are representing our specificy control ***rules as facts*** and leaving CLIPS rules, which can be called **meta-rules**, to define a *more general control strategy*.
For instance, in a diagnostic system, if we have pieces of knowledge like this one:

If *the patient has symptom A and symptom B* ***then*** *he can possibly have disease C.*

In our system we will encode them as some templated facts representing the LHS and RHS, then our rules could be something like:

If *there is a symptom that can be asked that can lead to a disease we could diagnose,* ***then*** *ask about that symptom*

We will show how to write meta-rules for modeling a backtracking strategy or to implement a decision tree searching strategy.

Decision Trees

Tree representation I

Following the animal classification example[2], suppose that we possess this pieces of knowledge:

If *the animal is not warm-blooded* ***then*** *it is a snake.*
If *the animal is warm-blooded and does not purr,* ***then*** *it is a dog.*
If *the animal is warm-blooded and does purr,* ***then*** *it is a cat.*

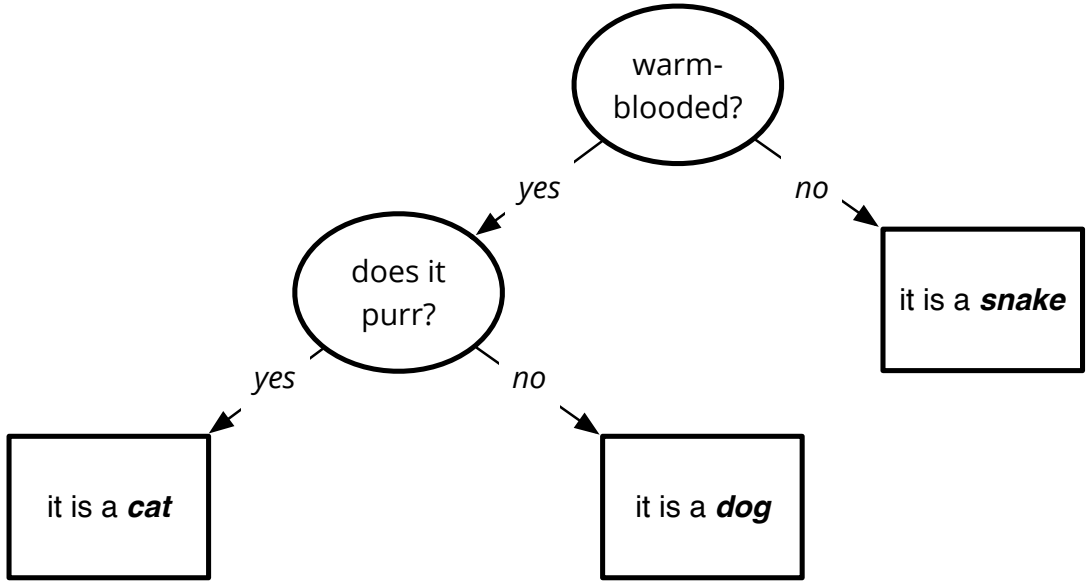
We will use decision trees as simple classifiers, expecting their inner nodes to be representing a choice to be answered (questions to the users); while leaves are the classification decision output (the animal race).

Instead of writing three CLIPS rules we will represent them with facts as a decision tree specifying its nodes and branches.
A decision tree encodes a set of rules by compactly representing the patterns that appear in their LHSs.

---

[2] See Giarratano's & Riley's book, chapter 12 and the provided code.

Tree representation II: example



Each rule is represented as a path going from the root node to a possible leaf node. Note that we need to start from a root. What about rules without any pattern in common?

It is also possible to have decision nodes with more outgoing branches, i.e. representing multiple choices.

Tree representation III

Nodes can be encoded as unordered facts with slots specifying the type (decision or answer) and branch type (yes-node no-node) and the eventual question or answer[3]:

```
(deftemplate node
    (slot name) (slot type)
    (slot question) (slot yes-node)
    (slot no-node) (slot answer))
```

The first node shall be the one named **root**.
Branches are expressed by saving the yes and no child node names.

Additionally, a fact **current-node** may be used to keep track of the node we are currently exploring (by saving its name).

[3] This representation may be redundant, can you simplify it?

Knowledge Base

In this representation, the KB will contain such nodes for our tree:

```
1  (node (name root) (type decision)  (question "Is the animal warm-blooded?")
2        (yes-node node1) (no-node node2) (answer nil))
3  (node (name node1) (type decision) (question "Does the animal purr?")
4        (yes-node node3) (no-node node4) (answer nil))
5  (node (name node2) (type answer) (question nil)
6        (yes-node nil) (no-node nil) (answer snake))
7  (node (name node3) (type answer) (question nil)
8        (yes-node nil) (no-node nil) (answer cat))
9  (node (name node4) (type answer) (question nil)
10       (yes-node nil) (no-node nil) (answer dog))
```

If we put it into a file, "ANIMAL.DAT", this would be the starting rule:

```
(defrule initialize
    (not (node (name root)))
    =>
    (load-facts "ANIMAL.DAT") (assert (current-node root)))
```

Visiting the tree I

Starting from the root the search advances by properly setting the current-node fact. In the case it is a decision node:

```
(defrule ask-decision-node-question
    ?node <- (current-node ?name)
    (node (name ?name) (type decision) (question ?question))
    (not (answer ?))
    =>
    (printout t ?question " (yes or no) ")
    (assert (answer (read))))
```

To move onto the yes or no branches, we could simply use two rules:

```
(defrule proceed-to-yes-branch
    ?node <- (current-node ?name)
    (node (name ?name) (type decision) (yes-node ?yes-branch))
    ?answer <- (answer yes)
    =>
    (retract ?node ?answer)
    (assert (current-node ?yes-branch)))
```

Visiting the tree II

Whether we reach a leaf, the user can be asked if the answer is correct

```
(defrule ask-if-answer-node-is-correct
    ?node <- (current-node ?name)
    (node (name ?name) (type answer) (answer ?value))
    (not (answer ?))
    =>
    (printout t "I guess it is a " ?value crlf)
    (printout t "Am I correct? (yes or no) ")
    (assert (answer (read)))) ;; we need to check the answer now
```

Supposing the answer was correct, one can restart the whole process by resetting the current-node to the root:

```
(defrule one-more-time
    ?phase <- (ask-try-again) ;; we can assert it at the end
    ?answer <- (answer yes)
    =>
    (retract ?phase ?answer)
    (assert (current-node root)))
```

Growing the tree I

Supposing the answer is incorrect, we could ask the user for the correct animal race and try to extend the tree. This corresponds to modify one rule in our KB.

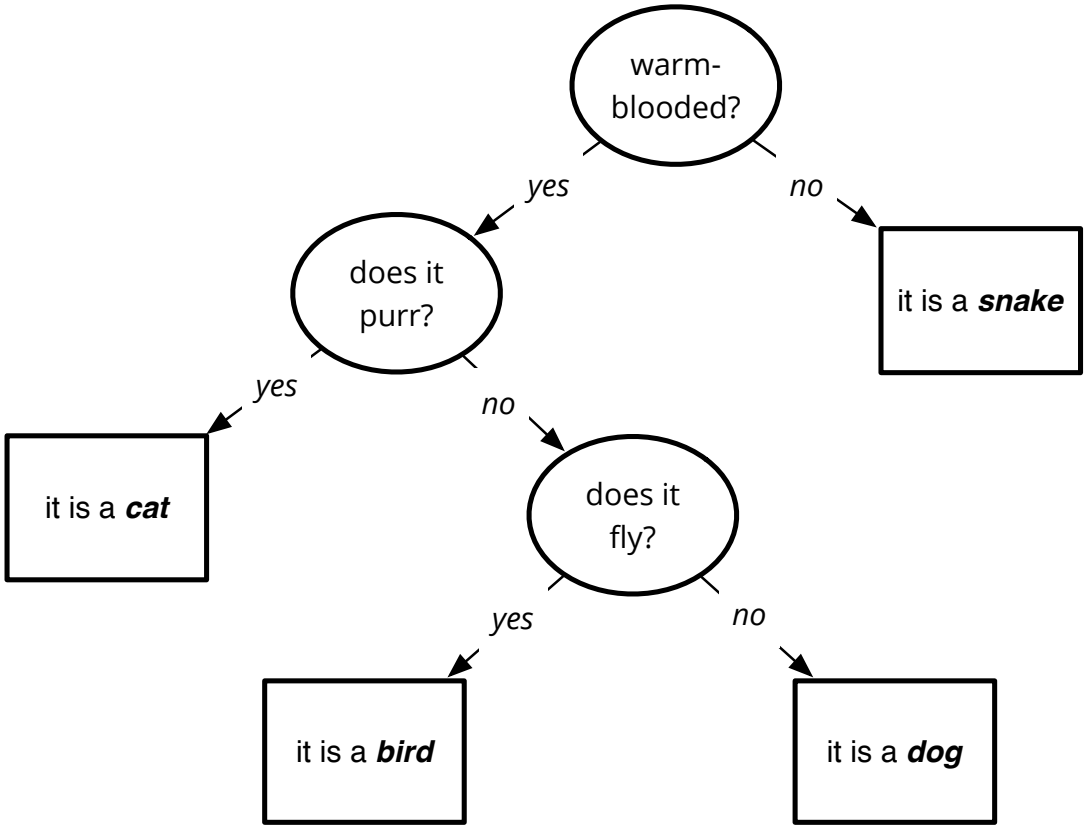
```
(defrule answer-node-guess-is-incorrect
    ?node <- (current-node ?name)
    (node (name ?name) (type answer))
    ?answer <- (answer no)
    =>
    (assert (replace-answer-node ?name))
    (retract ?answer ?node))
```

The function replace-answer-node has to substitute the old leaf with a subtree rooted in a decision node standing for a user supplied question, leading to a new yes branch (the correct answer) and a no branch (the old leaf).

Growing the tree II

```
CLIPS> (reset)
CLIPS> (run)
Is the animal warm-blooded? (yes or no) yes
Does the animal purr? (yes or no) no
I guess it is a dog
Am I correct? (yes or no) no
What is the animal? bird
What question when answered yes will distinguish
a bird from a dog? Does the animal fly?
Now I can guess bird
Try again? (yes or no) yes
Is the animal warm-blooded? (yes or no) yes
Does the animal purr? (yes or no) no
Does the animal fly? (yes or no) yes
I guess it is a bird
Am I correct? (yes or no) yes
Try again? (yes or no) no
CLIPS>
```

Growing the tree III



Decision trees: exercises

Make the original code more general and totally independent from the animal classification domain. Changing the KB containing the rules as the decision tree is easy. For instance, think about a simple decision tree to classify movies by genre.

Implement the rules presented in a simple diagnostic classifier as a decision tree. Note how the choice of a decision node position can change the tree structure but not the classification.
Can you implement a procedure (in Java for example) to random build the tree?

Extend the code to represent multiple choices decision nodes.

Backward chaining

A simple goal driven search

To implement a very simple backward chaining strategy, we will assume these simplifications[4]:

- facts and patterns are single valued attributes (without unknown values)
- LHSs are conjunctions of such patterns
- to match a pattern we check for the equality of its value
- goals (and RHS) are composed by single patterns
- if a goal cannot be derived by rules, it is asked to the user

Since CLIPS inference engine implements forward chaining, a backward chaining system in CLIPS would need a non banal representation for rules and facts.
The simplest thing is to use facts to represent and manipulate our KB rules, and defining meta rules for solving a goal by backward chaining.

[4] These examples and code are taken from Giarratano's and Riley's book, chapter 12.4.

Backward chaining: representation

The module BC will contains our representations.
Rules are unordered facts with multislots for the LHS and RHS part.

```
(deftemplate BC::rule
    (multislot if) (multislot then))
```

Matching is done by extracting from their lists the patterns, which are in the form of attribute **is** value. If more than one pattern is present in a LHS, then they are separated by the keyword **and**.
This is an example for a rule as a fact from the animal domain:

```
(rule (if warm-blooded is yes and does-purr is yes)
      (then animal is cat))
```

In the same module we can define templates for facts keeping track of the facts as attributes involved in matching and the current goal:

```
(deftemplate BC::attribute
    (slot name) (slot value))
(deftemplate BC::goal (slot attribute))
```

Wine suggester I

Suppose we are modeling the knowledge for a wine suggester that provides the best wine color to be associated with a dish, in the form of these rules:

If *the main course is red meat,* ***then*** *the best color is red*
If *the main course is fish* ***then*** *the best color is white*
If *the main course is poultry and it is turkey,* ***then*** *the best color is red*
If *the main course is poultry but it is not turkey,* ***then*** *the best color is white*

According to our conceptualization we would have:

```
(deffacts MAIN::wine-rules
    (rule (if main-course is red-meat) (then best-color is red))
    (rule (if main-course is fish) (then best-color is white))
    (rule (if main-course is poultry and meal-is-turkey is yes)
          (then best-color is red))
    (rule (if main-course is poultry and meal-is-turkey is no)
          (then best-color is white)))

(deffacts MAIN::initial-goal (goal (attribute best-color)))
```

Backward chaining: strategy I

The main strategy will revolve around trying to solve goals. We start with an input goal.

This is a sketch of the input procedure:

I. If a goal refers to a known asserted attribute, then the attribute value is returned.

II. Otherwise, we have to attempt to solve each rule whose RHS contains our current goal. For each rule, we try to solve recursively each pattern in the LHS, that is we set the current goal to them and repeat these steps again. If the value returned is not the one we searched for, we remove the rule from the WM, else we remove the attribute from the LHS and proceed.

III. If we were not able to satisfy the goal, we ask the user for the value, update the goal and return the value.

At each time, in the WM, we have the rules that we still can search for, the one that are unmatchable are removed. In the same way we remove the patterns in a LHS of a rule if they already matched some fact.

Backward chaining: strategy II

Supposing our WM contains the initial asserted rules for the wine suggester, if we happen to know that the main disch is poultry we can remove two rules since they are not applicable, ending up with just:

```
(rule (if main—course is poultry and meal—is—turkey is yes)
      (then best—color is red))
(rule (if main—course is poultry and meal—is—turkey is no)
      (then best—color is white)))
```

in the WM. Moreover, when we are testing for the remaining attributes in these rules LHSs, we can rewrite them as:

```
(rule (if meal—is—turkey is yes)
      (then best—color is red))
(rule (if meal—is—turkey is no)
      (then best—color is white)))
```

Note that in this way we are forgetting the initial formulation for our rules. This may be crucial if we wanted to implement an hypothesis testing approach.

Backward chaining: strategy III

While attempting a rule, if we match the RHS, we have to set a new goal for the first attribute in the LHS:

```
(defrule BC::attempt-rule
    (goal (attribute ?g-name))
    (rule (if ?a-name $?) (then ?g-name $?))
    (not (attribute (name ?a-name)))
    (not (goal (attribute ?a-name)))
    =>
    (assert (goal (attribute ?a-name))))
```

If no rule and attribute can satisfy a goal, let's ask the user, then assert it as a fact:

```
(defrule BC::ask-attribute-value
    ?goal <- (goal (attribute ?g-name))
    (not (attribute (name ?g-name)))
    (not (rule (then ?g-name $?)))
    =>
    (retract ?goal)
    (printout t "What is the value of " ?g-name "? ")
    (assert (attribute (name ?g-name)  (value (read)))))
```

Backward chaining: strategy IV

If a rule in the WM has a single pattern LHS and is satisfied, remove it and assert the matching fact:

```
(defrule BC::rule-satisfied
    (declare (salience 100))
    (goal (attribute ?g-name))
    (attribute (name ?a-name) (value ?a-value))
    ?rule <- (rule (if ?a-name is ?a-value)
                   (then ?g-name is ?g-value))
    =>
    (retract ?rule)
    (assert (attribute (name ?g-name) (value ?g-value))))
```

If we have a matching fact for our goal, remove the goal from the WM:

```
(defrule BC::goal-satisfied
    (declare (salience 100))
    ?goal <- (goal (attribute ?g-name))
    (attribute (name ?g-name))
    =>
    (retract ?goal))
```

Backward chaining: strategy V

If the rule is not applicable, just retract ii from the WM:

```
(defrule BC::remove_rule_no_match
    (declare (salience 100))
    (goal (attribute ?g_name))
    (attribute (name ?a_name) (value ?a_value))
    ?rule <- (rule (if ?a_name is ~?a_value)
                   (then ?g_name is ?g_value))
    =>
    (retract ?rule))
```

If one[5] of its LHS pattern, match, just remove it (and update the rule):

```
(defrule BC::modify_rule_match
    (declare (salience 100))
    (goal (attribute ?g_name))
    (attribute (name ?a_name) (value ?a_value))
    ?rule <- (rule (if ?a_name is ?a_value and $?rest_if)
                   (then ?g_name is ?g_value))
    =>
    (retract ?rule)
    (modify ?rule (if $?rest_if)))
```

[5] In this case always the first is checked, can you make match checking more flexible?

Wine suggester II

Here is a log with the the watch facts and watch rules commands, showing what is happening in the WM:

```
1  CLIPS> (reset)
2  <== f-0     (initial-fact)
3  <== f-7     (attribute (name main-course) (value poultry))
4  <== f-8     (rule (if meal-is-turkey is no) (then best-color is white))
5  <== f-11    (attribute (name meal-is-turkey) (value yes))
6  <== f-12    (attribute (name best-color) (value red))
7  ==> f-0     (initial-fact)
8  ==> f-1     (rule (if main-course is red-meat) (then best-color is red))
9  ==> f-2     (rule (if main-course is fish) (then best-color is white))
10 ==> f-3     (rule (if main-course is poultry and meal-is-turkey is yes) (then best-color is red))
11 ==> f-4     (rule (if main-course is poultry and meal-is-turkey is no) (then best-color is white))
12 ==> f-5     (goal (attribute best-color))
13 CLIPS> (run)
14 FIRE     1 start-BC: f-0
15 FIRE     2 attempt-rule: f-5,f-4,,
16 ==> f-6     (goal (attribute main-course))
17 FIRE     3 ask-attribute-value: f-6,,
18 <== f-6     (goal (attribute main-course))
```

Wine suggester III

```
19  What is the value of main-course? poultry
20  ==> f-7     (attribute (name main-course) (value poultry))
21  FIRE    4 remove-rule-no-match: f-5,f-7,f-2
22  <== f-2     (rule (if main-course is fish) (then best-color is white))
23  FIRE    5 remove-rule-no-match: f-5,f-7,f-1
24  <== f-1     (rule (if main-course is red-meat) (then best-color is red))
25  FIRE    6 modify-rule-match: f-5,f-7,f-4
26  <== f-4     (rule (if main-course is poultry and meal-is-turkey is no) (then best-color is white))
27  ==> f-8     (rule (if meal-is-turkey is no) (then best-color is white))
28  FIRE    7 modify-rule-match: f-5,f-7,f-3
29  <== f-3     (rule (if main-course is poultry and meal-is-turkey is yes) (then best-color is red))
30  ==> f-9     (rule (if meal-is-turkey is yes) (then best-color is red))
31  FIRE    8 attempt-rule: f-5,f-9,,
32  ==> f-10    (goal (attribute meal-is-turkey))
33  FIRE    9 ask-attribute-value: f-10,,
34  <== f-10    (goal (attribute meal-is-turkey))
```

Wine suggester IV

```
35  What is the value of meal-is-turkey? yes
36  ==> f-11    (attribute (name meal-is-turkey) (value yes))
37  FIRE    10 rule-satisfied: f-5,f-11,f-9
38  <== f-9     (rule (if meal-is-turkey is yes) (then best-color is red))
39  ==> f-12    (attribute (name best-color) (value red))
40  FIRE    11 goal-satisfied: f-5,f-12
41  <== f-5     (goal (attribute best-color))
42  CLIPS> (facts *)
43  f-0     (initial-fact)
44  f-7     (attribute (name main-course) (value poultry))
45  f-8     (rule (if meal-is-turkey is no) (then best-color is white))
46  f-11    (attribute (name meal-is-turkey) (value yes))
47  f-12    (attribute (name best-color) (value red))
48  For a total of 5 facts.
```

Backward chaining: exercises