Production Systems

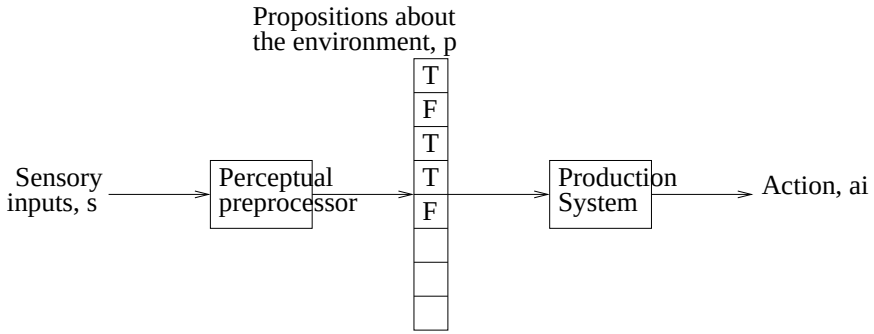
The table in a table-driven agent implements the agent's action function. But this way of implementing an action function is far from concise. There is an entry in the table for every possible percept (and, in the case of Q-Learning, an entry for every percept-action pair). The wall-following softbot, for example, has eight touch sensors, each returning only 0 or 1, and hence 256 entries in its table. With more sensors, and sensors that can return more values than just 0 or 1, this quickly becomes impractical (the table becomes too large) or even infeasible (the table is infinite). For example, a light sensor or a thermometer returns a real value; unless we both restrict the range of values and discretise them, so that the light sensor or thermometer can return only one of a finite set of values (and preferably one of a *small* set), we cannot build a table-driven agent that has these sensors. In this lecture, we are not going to deal directly with the problem of real-valued percepts. But we are going to see a way of building agents which implement their action functions in a way that is more concise than using a table. We continue to focus on reactive agents.

1 Building Reactive Agents using Production Systems

One way of building such an agent is to use a *production system*.

Such an agent has a vector of sensory inputs $\mathbf{s} = \langle s_1, \ldots, s_n \rangle$. It applies some preprocessing to these to produce a vector $\mathbf{p} = \langle p_1, \ldots, p_i, \ldots, p_m \rangle$ of propositions. For the moment, we will assume that each p_i is either T (true) or F (false). Each p_i represents some simple proposition about the environment, e.g. that there is an obstacle directly in front of the agent; that the agent is in a corner; that the temperature is greater than 22^oC; that the amount of light entering a light meter exceeds a threshold value; etc. Finally, the agent has a vector of actions that it can choose to execute, $\mathbf{a} = \langle a_1, \ldots, a_l \rangle$.

The agent then needs a way of implementing its action function. This is where we use the production system. It will choose which action to execute on the basis of vector $\mathbf{p}$.



A production system comprises a list of *condition-action rules*. Elsewhere, condition-action rules might be referred to as: production rules, productions, situation-action rules, stimulus-response rules or just rules. (We will explain below that these are not the same kind of rules that we saw in rule-based classifiers, or in rule-based reasoning in general.)

Each condition-action rule is of the form

if c **then** a_i

where c is a *condition* and a_i is one of the agent's possible actions. The conditions will be Boolean-functions of the propositions in $\mathbf{p}$, e.g. using conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), etc.

The agent will execute the action of one of the rules whose condition evaluates to T (true). Of course, in general, several rules will have true conditions. We refer to these as the *conflict set*. We will need a *conflict resolution strategy* to pick one of these rules.

In pseudocode, the agent does the following at each point in time:

```
s := SENSE();
p := PREPROCESS(s);
conflictSet := FINDALLMATCHES(p, rules);
rule := CHOOSEARULE(conflictSet);
action := the action part of rule;
EXECUTE(action);
```

There are many possible conflict resolution strategies, including:

- Choose the *first* rule that matches.
- Assign priorities to the rules. Choose the rule from the conflict set that has *highest priority*.
- Choose the *most specific* rule. Assuming rule conditions are conjunctions of propositions, this is the rule in the conflict set that has the longest condition.
- Choose a rule that has not been used before or the one that has been used *least recently*.

We will only consider production systems where the conflict resolution strategy is to execute the action of the *first* rule whose condition is true. The pseudocode from above can therefore be simplified:

```
s := SENSE();
p := PREPROCESS(s);
rule := FINDFIRSTMATCH(p, rules);
action := the action part of rule;
EXECUTE(action);
```

2 Examples of Production System

2.1 Example A

By way of a simple example, consider a robot who lives in a room in which there is a moving light source. The robot has two light sensors on its front: a left sensor and a right sensor. Each of these can return a real number to signify the amount of light being received. The robot is capable of three actions: `Move` (move one pace forward), `Turn(RIGHT, 1)` (turn 45^o to the right) and `Turn(LEFT, 1)` (turn 45^o to the left).

So, vector $\mathbf{s}$ is of length two: s_0 says how much light the left sensor is receiving; s_1 says how much light the right sensor is receiving.

Vector $\mathbf{p}$ is also of length two. p_0 will be T iff $s_0 > s_1$; p_1 will be T iff $s_1 > s_0$. (If p_0 and p_1 are both F, then the two sensors must be receiving the same amount of light.)

The production system is:

if p_0 **then** `Turn(LEFT, 1)`
if p_1 **then** `Turn(RIGHT, 1)`
if $\neg p_0 \wedge \neg p_1$ **then** `Move`

Class Exercise: *What will be the behaviour of this robot?*

2.2 Example B

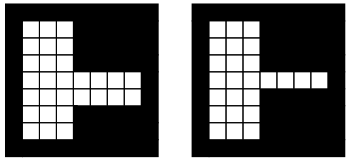
Consider the softbot that walks anticlockwise around the walls of grid-based rooms. It has 8 sensors mounted around its body; we will use 8 propositions, $p_0, \ldots, p_7$, where p_i will be T iff s_i returns 1 (is touching an object). One possible production system is:

if $\neg p_2 \wedge p_3$ **then** `Turn(RIGHT, 2)`
if p_0 **then** `Turn(LEFT, 2)`
if $\neg p_0$ **then** `Move`

Note how the ordering of the rules is important. Note also how much more concise this is than the original table-driven agent.

Class Exercise. This agent is unable to handle 'tight dead-end corridors', i.e. a dead-end path between two walls that is less than two cells wide, e.g.:

Our agent can cope with this... but not with this.



Indeed, no purely reactive agent can handle these. Why not?

3 Default Actions in Production Systems

Suppose you have a production system whose conflict resolution strategy is to always choose the first-matching rule. When devising the rules for your production system, it is conventional for the last rule to have as its condition simply true (the proposition that is always T):

if true **then** a_{default}

So if no other rule (higher in the list) is used, this rule will be used as a last resort. The action in this rule, a_{default}, will be some suitable default action to be executed when nothing else is suitable. This may prevent your agent from being paralysed by inaction.

This simple change works well for Example B above: the third rule could be rewritten as **if** true **then** `Move`.

Class Exercise: Consider Example A from above.

1. Under what circumstances did this agent 'freeze'?
2. How would you rewrite the rules so that the agent can get itself out of these difficulties?

4 Agents with Goals

At the moment, the example agents that we have designed (the table-driven agents and the production system agents) are always 'on the go'. At every moment of time, they sense, plan and act. Suppose instead you want your agent to achieve some goal and then stop.

Class Exercise: How would you write your production system to achieve this?

5 Comparison with Propositional Definite Clauses

Recall that propositional definite clauses can be either propositional symbols on their own (called facts), e.g.

$$p$$

or a conditional whose antecedent is a conjunction of propositional symbols and whose consequent is a single propositional symbol (called rules), e.g.

$$(p_1 \wedge p_2 \wedge \ldots \wedge p_n) \Rightarrow q$$

Condition-action rules and definite clauses are utterly different beasts. What are the differences?

- We restricted the antecedent of definite clauses to a conjunction of propositional symbols: no negation, no disjunction, etc. Lifting this restriction would have far-reaching effects on the forwards- and backwards-chaining inference engines. We have placed no such restriction on the 'if' parts of condition-action rules.
- The biggest difference: In condition-action rules, the 'then' part of the rule is an *action* to be executed; in definite clauses, the 'then' part (the consequent) is a statement (a single propositional symbol). Numerous things follow from this:
  - Propositional definite clauses make statements and are hence true or false; condition-action rules are commands and hence not true or false.
  - Propositional definite clauses 'chain' together: the consequent of one may be in the antecedent of another. Condition-action rules do not chain together: the 'then' part of one condition-action rule can't appear in the 'if' part of another.

Or are they so different? Suppose we are writing a production system, comprising condition-action rules, but our agent has an `Assert` action, which stores whatever is asserted into some area of the agent's memory. Here's an example:

if *suckles* **then** `Assert(`*`mammal`*`)`
if *hairy* **then** `Assert(`*`mammal`*`)`
if *mammal* $\wedge$ *chewsCud* **then** `Assert(`*`ungulate`*`)`
if *ungulate* $\wedge$ *longNeck* **then** `Assert(`*`giraffe`*`)`
...

Now the differences have been eroded.

A richer repertoire of actions also means that we are not confined to implementing reactive agents. If the agent has an `Assert` action, for example, then it can remember its percepts and use them in future decisions. It is also possible for it to simulate actions in its head (i.e. to think ahead). Indeed, highly sophisticated agents have been built using production systems where the agent is equipped with actions such as `Assert` and `Retract`.

6 Genetic Algorithms for Production Systems

It's possible to learn a production system, taking an approach similar to Q-Learning. We won't look at the details. Instead, we'll consider evolving the production system.

Suppose we have m propositions, $p_0, \ldots, p_m$ and l actions $a_0, \ldots, a_l$. Each condition-action rule can be represented by a bit string comprising m bits to represent the rule's condition and $\lceil \log_2 l \rceil$ bits to represent the rule's action. For example, if we have 8 propositions ($m = 8$), we need 8 bits for the rule's condition. And if we have 4 actions then we need 2 bits for the rule's action. As before, each of the agent's possible actions is assigned a unique bit string, e.g.:

Action	Bit string
`Move`	00
`Turn(RIGHT, 2)`	01
`Turn(RIGHT, 4)`	10
`Turn(LEFT, 2)`	11

Now consider the following rule:

if p_0 **then** `Turn(LEFT, 2)`

The condition of this rule can be represented as the following bit string 1#######000. In these bit strings, we use 0 (false) and 1 (true), as you'd expect, but we also allow # to represent Don't Care (so they're not really bit strings). So the string 1####### says that p_0 must be true and we don't care about the other propositions. Given that `Turn(LEFT, 2)` is encoded by 11 (above), the rule as a whole is represented as

$$1\#\#\#\#\#\#\#11$$

The rule

if p_1 **then** `Turn(RIGHT, 2)`

is similarly represented by

$$\#1\#\#\#\#\#\#01$$

The rule

if $\neg p_0 \wedge \neg p_1$ **then** `Move`

is represented by

$$00\#\#\#\#\#\#00$$

Class Exercise. What would be the bit string representation of the following rule?

if $p_3 \wedge p_4 \wedge \neg p_0 \wedge p_7$ **then** `Move`

In the previous sections of these notes, we have allowed the rules' conditions to use any Boolean operators in any combination. But the bit string representation we have devised above is not as expressive. Mathematically speaking, this representation allows only rule conditions that are in *conjunctive normal form*, i.e. they comprise proposition symbols or the negation of proposition symbols conjoined (ANDed) together. It does not allow, for example, disjunction in rule conditions. But, as we have seen before, simple cases of disjunction can be had through multiple rules. For example, the rule:

if $p_0 \vee p_1$ **then** `Move`

can be equivalently represented by two rules:

if p_0 **then** `Move`

if p_1 **then** `Move`

Having described how to encode individual rules, we must now decide how to encode the whole production system. Quite simply, the production system as a whole is encoded by concatenating the bit strings for each rule.

This raises an issue. Different production systems will be encoded by different length bit strings. For example, a production system with 3 rules will have a bit string of length 30 (because in our example each rule is encoded by a bit string of length 10); but a production system with 4 rules will have a bit string of length 40. Either crossover and mutation must be carefully defined so that they gracefully handle different-length parents; alternatively, we could 'pad out' all strings so that they are all of the same length.

The great advantage of bit strings is they give easy implementation of genetic operators. However, our very efficient implementations, using logical operations and masks, cannot work on strings that allow # as well as 0 and 1. Either we must abandon those fast implementations, or we must find a way of avoiding use of #.

Class Exercise. Give an encoding that avoids #.