AITA : Production Systems

1. What is a Production System?
2. Inference Rules
3. Recognize-Act Cycle
4. Matching and Binding
5. Forward Chaining
6. Backward Chaining
7. Problems with Recognize-Act Cycle
8. Reason Maintenance Systems
9. Forward or Backward Chaining
10. Conflict Resolution

What is a Production System?

A ***production system*** consists of four basic components:

1. A ***set of rules*** of the form $C_i \rightarrow A_i$ where C_i is the condition part and A_i is the action part. The condition determines when a given rule is applied, and the action determines how it is applied.
2. One or more ***knowledge databases*** that contain whatever information is relevant for the given problem. Some parts of the database may be permanent, while others may temporary and only exist during the solution of the current problem. The information in the databases may be structured in any appropriate manner.
3. A ***control strategy*** that determines the order in which the rules are applied to the database, and provides a way of resolving any conflicts that can arise when several rules match at once.
4. A ***rule applier*** which is the computational system that implements the control strategy and applies the rules.

We shall make these ideas more concrete over the course of the next four lectures.

Inference Rules

We are already familiar with the kind of rules our AI systems may use, e.g.

Deductive Inference Rule

Modus Ponens

Given "A" and "A implies B", we can conclude "B":

$$\frac{A \qquad A \Rightarrow B}{B}$$

Example:

$$\frac{\text{It is raining} \qquad \text{If it is raining, the street is wet}}{\text{The street is wet}}$$

Abductive Inference Rule

Abduction

Given "B" and "A implies B", it might be reasonable to expect "A":

$$\frac{B \qquad A \Rightarrow B}{A}$$

Example:

$$\frac{\text{The street is wet} \qquad \text{If it is raining, the street is wet}}{\text{It is raining}}$$

Recognize-Act Cycle

Typically, our production systems will have a rule interpreter that takes the form of a ***Recognize-Act Cycle***. This cycle has four stages:

1. ***Match*** the condition/premise patterns in the rules against the elements in the working memory to identify the set of applicable rules.
2. If there is more than one rule that can be 'fired' (i.e. that can be applied), then use a ***Conflict Resolution*** strategy to choose which one to apply. If no rules are applicable, then stop.
3. ***Apply*** the chosen rule, which may result in adding a new item to the working memory, or in deleting an old one.
4. ***Check*** if the terminating condition is fulfilled. If it is, then stop. Otherwise, return to stage 1.

The ***termination condition*** can either be defined by a goal state, or by a cycle condition (e.g. a maximum number of cycles).

Matching

The condition/premise patterns in the rules need to be ***matched*** with the known facts.

Consider, a typical rule (about the value of horses) that matches a set of facts:

Rule

IF:	*x* is a horse
	x is the parent of *y*
	y is fast
THEN:	*x* is valuable

Facts

Comet	is-a	horse
Prancer	is-a	horse
Comet	is-parent-of	Dasher
Comet	is-parent-of	Prancer
Prancer	is	fast
Dasher	is-parent-of	Thunder
Thunder	is	fast
Thunder	is-a	horse
Dasher	is-a	horse

In general there will be ***variables*** (e.g. *x* and *y*) in the rules which stand for arbitrary objects. We need to find ***bindings*** for them so that the rule is applicable.

Binding

We simply need to see which values can be assigned to the variables in the rule. For example, "Comet is-a horse" matches "*x* is-a horse", but "Comet is-a lion" would not.

Bindings for "*x* is-a horse"	**Bindings for "*y* is fast"**	**Bindings for "*x* is a parent of *y*"**
x = Comet	*y* = Prancer	*x* = Comet , *y* = Dasher
x = Prancer	*y* = Thunder	*x* = Comet , *y* = Prancer
x = Thunder		*x* = Dasher , *y* = Thunder
x = Dasher		

From these we can deduce that there are two possible bindings applicable to the rule:

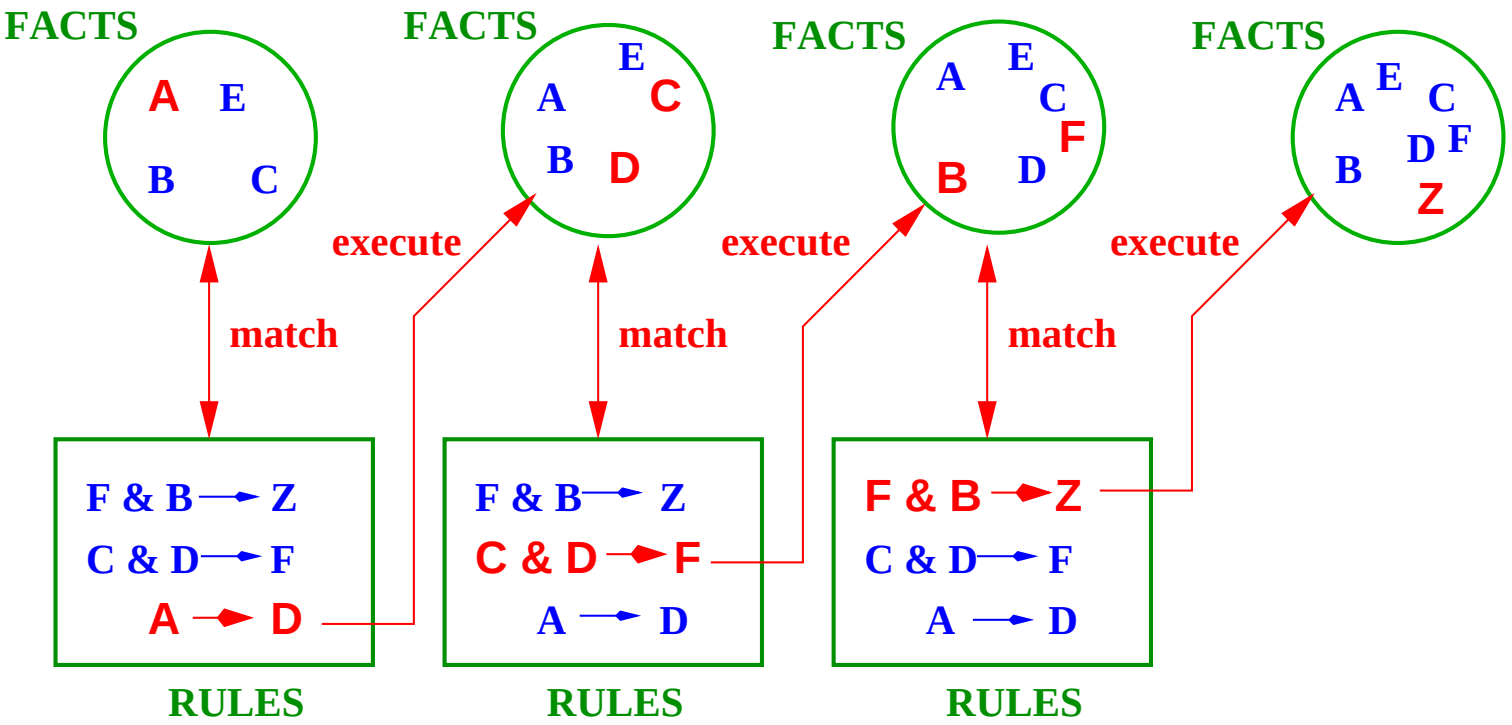
"*x* = Comet and *y* = Prancer" , "*x* = Dasher and *y* = Thunder"

The rule then tells us "*x* is valuable", i.e. "Comet is valuable" and "Dasher is valuable".

Forward Chaining

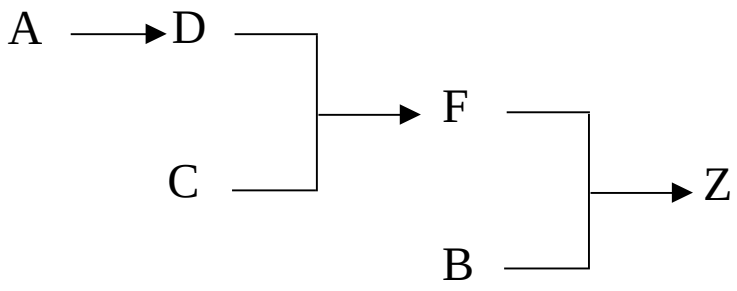
Forward chaining or ***data-driven inference*** works by repeatedly: starting from the current state, matching the premises of the rules (the IF parts), and performing the corresponding actions (the THEN parts) that usually update the knowledge base or working memory.

The process continues until no more rules can be applied, or some cycle limit is met, e.g.



The Forward Inference Chain

In this example there are no more rules, so we can draw the inference chain:



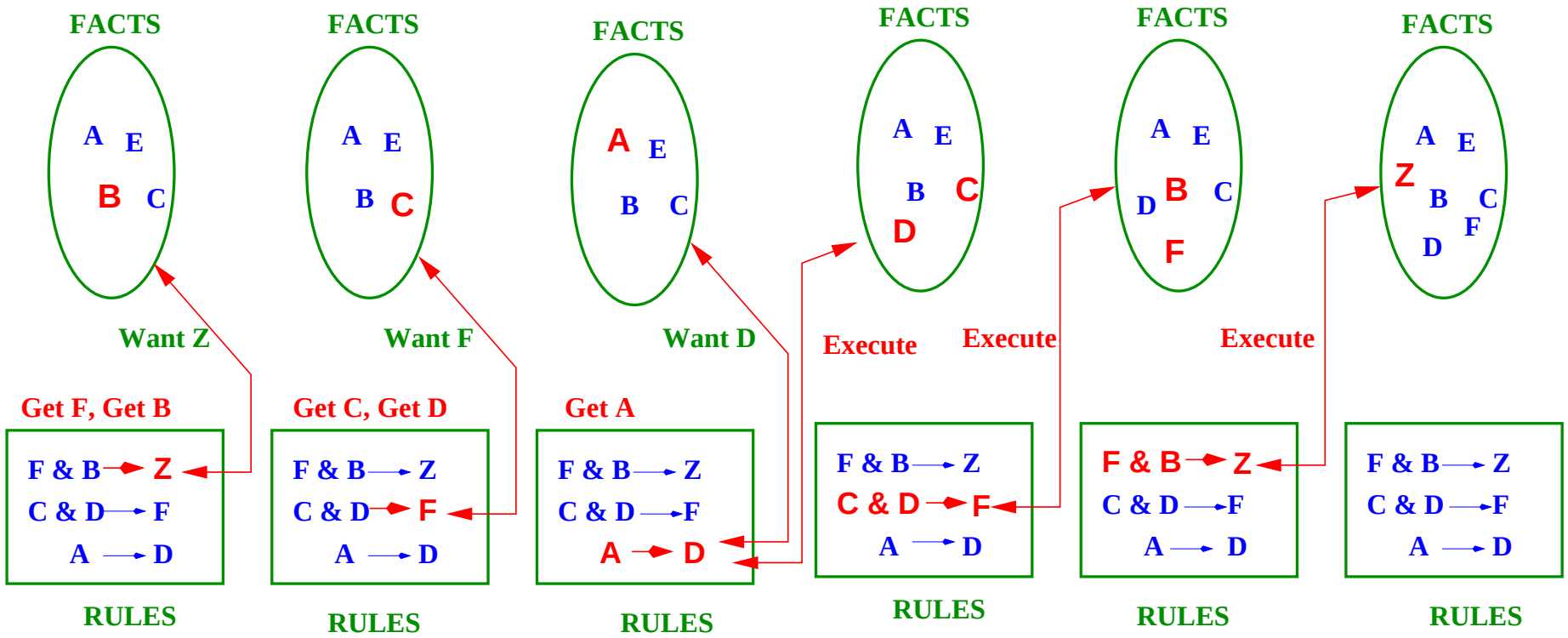
This seems simple enough, but in this case we only had a few initial facts, and a few rules. Generally, things will not be so straight forward.

Disadvantages of Forward Chaining

1. Many rules may be applicable at each stage – so how should we choose which one to apply next at each stage?
2. The whole process is not directed towards a goal, so how do we know when to stop applying the rules?

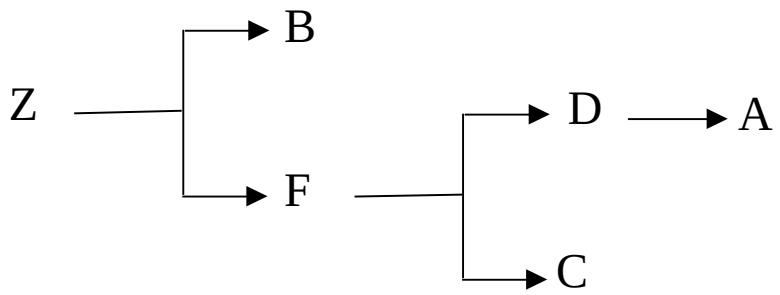
Backward Chaining

Backward chaining or ***goal-driven inference*** works towards a final state by looking at the working memory to see if the ***sub-goal*** states already exist there. If not, the actions (the THEN parts) of the rules that will establish the sub-goals are identified, and new sub-goals are set up for achieving the premises of those rules (the IF parts). The previous example now becomes:



The Backward Inference Chain

The first part of the chain works back from the goal until only the initial facts are required, at which point we know how to traverse the chain to achieve the goal state.



Advantage of Backward Chaining

1. The search is goal directed, so we only apply the rules that are necessary to achieve the goal.

Disadvantage of Backward Chaining

1. A goal has to be known. Fortunately, most AI systems we are interested in can be formulated in a goal based fashion.

Problems with the Recognize-Act Cycle

We can identify four important potential problems with the Recognize-Act Cycle:

1. How do we know that our production system is ***Consistent***?

   $\Rightarrow$ We need a ***Reason Maintenance System***

2. How do we know which ***Global Strategy*** for rule choice to apply?

   $\Rightarrow$ We need to choose between ***Forward and Backward*** Chaining

3. How do we know which ***Local Strategy*** for rule choice to apply?

   $\Rightarrow$ We need a ***Conflict Resolution System***

4. How do we Maximize ***Efficiency*** as the Complexity increases?

   $\Rightarrow$ We can use the ***Rete Algorithm***

We shall study the first three points in the remainder of this lecture, and cover the Rete Algorithm when we get to the lecture on Expert System Building.

Reason Maintenance System

Recall the "Raining example" that we looked at before. That included the rule:

$$\text{IF} \quad \text{Raining} \wedge \text{Outside} \wedge \neg\text{HasUmbrella} \quad \text{THEN} \quad \text{Wet}$$

Now suppose we have just gone outside and have the initial facts:

$$\text{Dry}, \quad \text{Outside}, \quad \text{Raining}$$

We can use the rule to generate a new fact, leaving us with:

$$\text{Dry}, \quad \text{Outside}, \quad \text{Raining}, \quad \text{Wet}$$

Which is now an inconsistent set of facts. A full scale system will have 1000s of rules and facts with much scope for inconsistencies. We need to build some kind of ***Reason Maintenance System*** into our system to deal with the inconsistencies. The literature contains many examples of how these work (e.g. Nilsson, section 17.3).

Forward or Backward Reasoning?

There are four major factors to help us choose between forward and backward reasoning:

1. **Are there more possible start states or goal states?**

   In general it is best to move from the smaller set of states to the larger.

   (Consider the situation of travelling between home and an unfamiliar place – it makes sense to use forward chaining to get home from the unfamiliar place, but backward chaining to get from home to the unfamiliar place.)

2. **Do we require the program to justify its reasoning?**

   If so, we should prefer the direction that corresponds more closely with the way that users think.

   (It is a common requirement that AI systems must be able to justify their reasoning in terms that the user can easily understand. This is because many users will not trust an AI system that operates in a manner that they can't understand.)

3. **What kind of events trigger problem solving?**

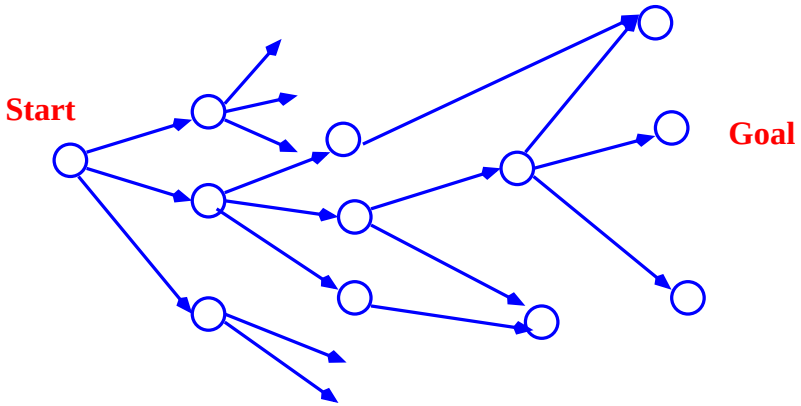
   If it is the arrival of a new fact, then forward chaining makes sense.

   If it is a query to which a response is required, then backward chaining will be more natural.

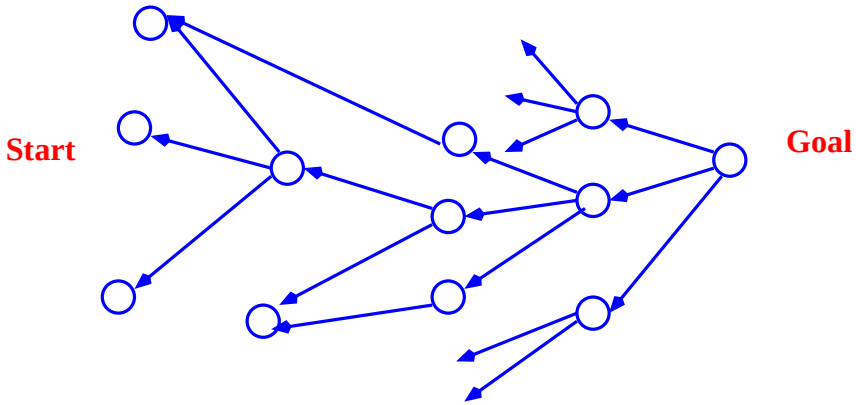
4. **In which direction is the branching factor greatest?**

   Go in the direction with the lower branching factor.

Backward Chaining Better



Forward Chaining Better



Conflict Resolution

Recall the four stages of the Recognise Act Cycle:

1. ***Match*** the rules against the known facts to see which rules can fire.
2. If more than one rule can fire, use a ***Conflict Resolution*** strategy to choose one.
3. ***Apply*** the chosen rule, updating the list of known facts.
4. Check the ***Termination Criterion*** and either stop or return to step 1.

Clearly, decisions made at the conflict resolution stage will be crucial since they can dramatically affect the solution, and how quickly we reach it.

The ***conflict set*** is defined as the set of pairs of the form:

⟨ Production rule, matching working memory elements ⟩

and we need to chose one. We need to distinguish between ***general conflict resolution*** and *pr**oblem specific conflict resolution***.

General Conflict Resolution Strategies

Perhaps the five most common ***General Conflict Resolution Strategies*** are:

1. Delete instantiations of rules that have already fired.
2. Order instantiations by the generation age of all the elements. Prefer the youngest.
3. Compare the generation age of the elements in working memory which match the first condition of the rules. Prefer the youngest.
4. Prefer the most specific rules (i.e. those with the most pre-conditions).
5. Random choice.

The ***rationale*** behind these strategies is fairly obvious:

1. Prevents obvious endless loops of the same rules being applied with the same facts.
2. New elements are more likely to describe the current situation.
3. As for strategy 2, but may be more efficient.
4. This catches any exceptions/special cases before applying more general rules.
5. Very easy to compute.

Specific Conflict Resolution Example

Consider the following set of rules:

R1:	IF:	engine does not turn AND battery is not flat
	THEN:	ask user to test starter motor
R2:	IF:	there is no spark
	THEN:	ask user to check the points
R3:	IF:	engine turns AND engine does not start
	THEN:	ask user to check the spark
R4:	IF:	engine does not turn
	THEN:	ask user to check the battery
R5:	IF:	battery is flat
	THEN:	ask user to charge battery AND EXIT

If the initial facts are “engine does not turn” and “battery is not flat”, the conflict set is:

{ ⟨ **R1**, engine does not turn, battery is not flat ⟩, ⟨ **R4**, engine does not turn ⟩ }

We can see that our general conflict resolution strategy 4 would work well here.

Problem Specific Conflict Resolution – Extra Conditions

The easiest way to proceed in problem specific cases is to simply ***add extra conditions*** to the rules to avoid the conflicts.

These extra conditions can be related to the inference strategies, e.g. to what is currently being searched for, or to what rule applications tend to be most useful.

In our previous example we might modify R1 to give:

R1: IF: *haven't already tested starter motor*
AND engine does not turn
AND battery is not flat
THEN: ask user to test starter motor

Disadvantages

1. We will end up with a mixture of heuristic and factual knowledge.
2. Large knowledge bases will not be easily maintainable.

Problem Specific Conflict Resolution – Meta-Rules

It makes sense avoid mixing the conflict resolution heuristics with the rule base by separating the object level knowledge from the ***meta-level knowledge***.

In our previous example we might supplement rule R1 with a ***meta-rule*** to give:

RULE R1:

IF: engine does not turn
AND battery is not flat
THEN: ask user to test starter motor

META-RULE M1:

IF: *haven't already tested starter motor*
THEN: *select R1*

Naturally this will increase the overall number of rules, but it does separate the factual and heuristic knowledge and makes the knowledge bases easier to maintain.

Overview and Reading

1. We began by defining the idea of a production system, recalling the basic types of inference rule, and specifying the basic recognize-act cycle.
2. We then looked at matching, binding, forward chaining, and backward chaining.
3. We ended looking at how to deal with inconsistencies, how to choose between forward and backward chaining, and how to perform conflict resolution.

Reading

1. Winston: Chapter 7
2. Jackson: Sections 5.1, 5.2, 5.3
3. Negnevitsky: Section 2.1, 2.2, 2.6, 2.8
4. Rich & Knight: Sections 2.2, 2.4, 7.2.1
5. Russell & Norvig: Sections 9.1, 9.2, 9.3, 9.4
6. Nilsson: Section 2.2.1, 17.3