Rule-Based Systems

Rule-Based Systems

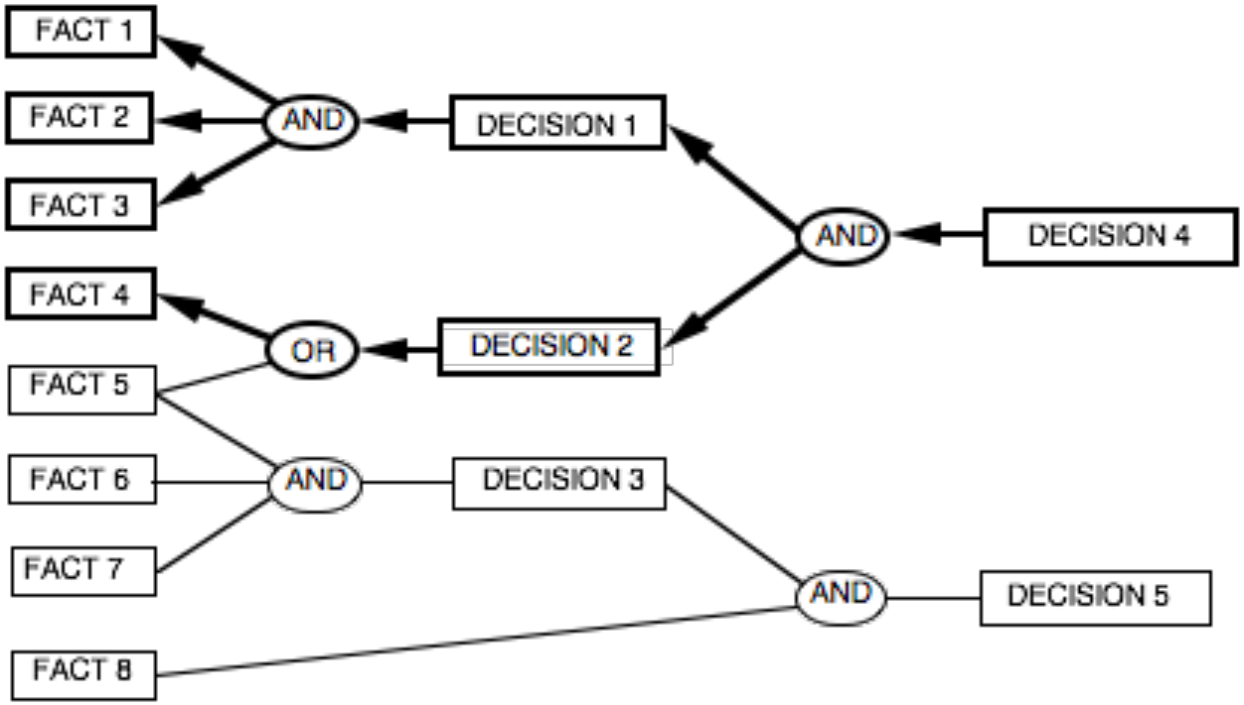
The components of a rule-based system have the form:

if <condition> then <conclusion>

Rules can be evaluated by:

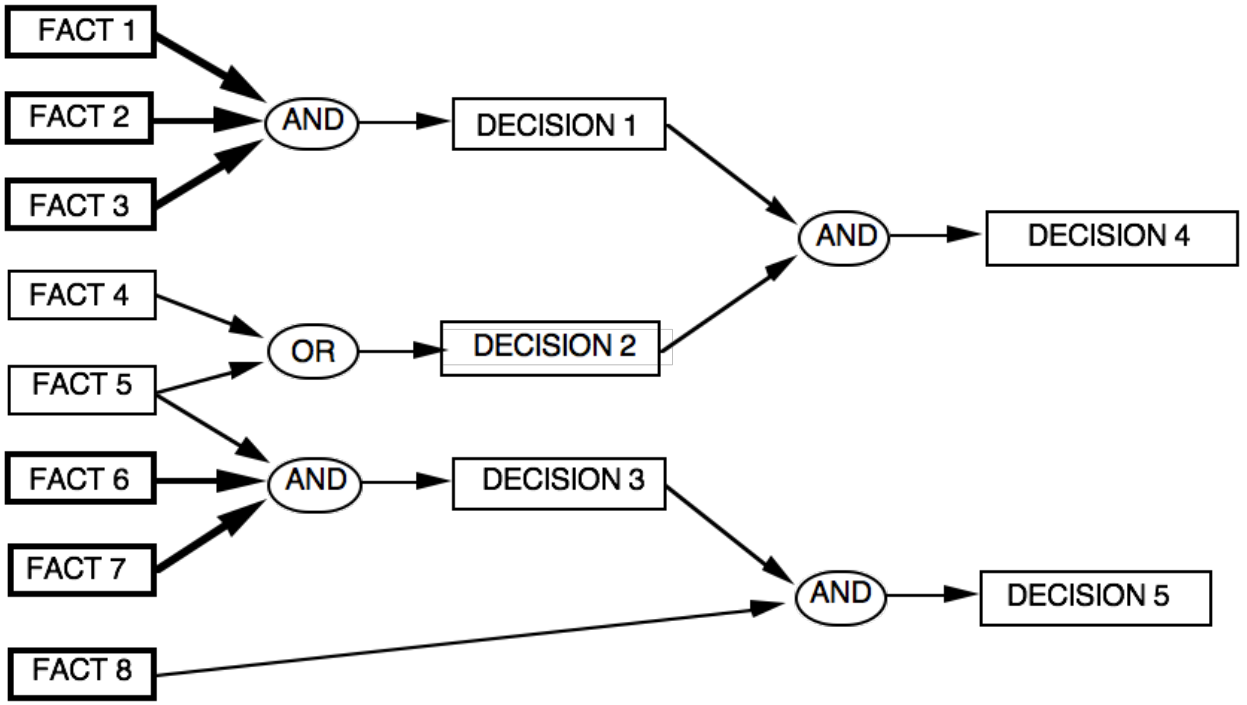
- backward chaining
- forward chaining

Backward Chaining



- To determine if a decision should be made, work backwards looking for justifications for the decision.
- Eventually, a decision must be justified by facts.

Forward Chaining



- Given some facts, work forward through inference net.
- Discovers what conclusions can be derived from data.

Forward Chaining

Until a problem is solved or no rule's 'if' part is satisfied by the current situation:

1. Collect rules whose 'if' parts are satisfied.

   If more than one rule's 'if' part is satisfied, use a conflict resolution strategy to eliminate all but one.

2. Do what the rule's 'then' part says to do.

Production Rules

A production rule system consists of

- a set of rules
- working memory that stores temporary data
- a forward chaining inference engine

Match-Resolve-Act Cycle

loop

match conditions of rules with contents of working memory

if no rule matches then stop

resolve conflicts

act (i.e. perform conclusion part of rule)

end loop

BAGGER

1. Check what the customer has selected.

   Look to see if something is missing, suggest additions.
2. Put the large items in the bag.

   Put big bottles first.
3. Put in the medium sized items.

   Put frozen food in plastic bags.
4. Put in the small items wherever there is room.

Working Memory

Step: Check order

Bag1: <empty>

Unpacked: Bread

Glop

Granola (2)

Ice cream

Chips

Attributes of Objects

ITEM	CONTAINER TYPE	SIZE	FROZEN?
Bread	Plastic bag	Medium	No
Glop	Jar	Small	No
Granola	Cardboard box	Large	No
Ice cream	cardboard carton	Medium	Yes
Pepsi	Bottle	Large	No
Chips	Plastic bag	Medium	No

Rules for Step 1

B1:

if the step is check-order

and there is a bag of chips

and there is no soft-drink bottle

then add one botle of soft drink to the order

B2:

if the step is check-order

then discontinue the check-order step

and start the pack-large-items step

Which of these rules should be chosen when in the check order step?

Conflict Resolution

Specificity Ordering

If a rule's condition part is a superset of another, use the first rule since it is more specialised for the current task.

Rule Ordering

Choose the first rule in the text, ordered top-to-bottom.

Data Ordering

Arrange the data in a priority list. Choose the rule that applies to data that have the highest priority.

Conflict Resolution

Size Ordering

Choose the rule that has the largest number of conditions.

Recency Ordering

The most recently used rule has highest priority

The least recently used rule has highest priority

The most recently used datum has highest priority

The least recently used datum has highest priority

Context Limiting

Reduce the likelihood of conflict by separating the rules into groups, only some of which are active at any one time. Have a procedure that activates and deactivates groups.

Rules for Step 2

B3:
if the step is pack-large-items
and there is a large item to be packed
and there is a large bottle to be packed
and there is a bag with < 6 large items
then put the bottle into the bag

B4:
if the step is pack-large-items
and there is a large item to be packed
and there is a bag with < 6 large items
then put the large item into the bag

B5:
if the step is pack-large-items
and there is a large item to be packed
then get a new bag

Working Memory So Far

Step:	pack-medium-items
Bag1:	Pepsi
	Granola (2)
Unpacked:	Bread
	Glop
	Ice cream
	Chips

Rules for Step 3

B7:
if the step is pack-medium-items
and there is a medium item to be packed
and there is an empty bag or a bag with medium items
and the bag is not yet full
and the medium item is frozen
and the medium item is not in a freezer bag
then put the medium item in a freezer bag

B8:
if the step is pack-medium-items
and there is a medium item to be packed
and there is an empty bag or a bag with medium items
and the bag is not yet full
then put the medium item in the bag

Rules for Step 3

B9:

if the step is pack-medium-items

and there is a medium item to be packed

then get a new bag

B10:

if the step is pack-medium-items

then discontinue the pack-medium-items step

and start the pack-small-items step

Working Memory So Far

Step:	pack-small-items
Bag1:	Pepsi
	Granola (2)
Bag2:	Bread
	Ice cream (in freezer bag)
	Chips
Unpacked:	Glop

Rules for Step 4

B11:

if the step is pack-small-items
and there is a small item to be packed
and the bag is not yet full
and the bag does not contain bottles
then put the small item in the bag

B12:

if the step is pack-small-items
and there is a small item to be packed
and the bag is not yet full
then put the small item in the bag

Rules for Step 4

B13:

if the step is pack-small-items

and there is a small item to be packed

then get a new bag

B14:

if the step is pack-small-items

then discontinue the pack-small-items step

and stop

Implementing Rules in Prolog

```
% Rules


rule1
if    a
and   b
and   c
then  d.


rule2
if    a
and   b
and   c
then  d.
```

- We want to express rules in a natural way.
- Prolog's allows you to define your own operators to make data structures readable
- a, b, c, etc may be any Prolog term, including variables

Defining Operators

```
op(900, xfx ,if)!
op(800, xfx, then)!
op(700, xfy, and)!
```

- First argument is the operator's precedence
- Second argument is the operator's associativity
  - yfx – left associative
  - xfy – right associative
  - xfx – non-associative
- Third argument is the operator's name (may be a list of names with the same properties).

Working Memory

```
% Working memory


wm(a).
wm(b).
wm(c).
```

- We use a predicate symbol, "wm", to distinguish working memory elements in Prolog's database

The Match-Resolve-Act Cycle

```
exec :-
    repeat,
    select_rule(R),
    fire(R), !.
```

Always succeeds on backtracking

- If "fire" succeeds, cut will prevent backtracking
- If "fire" fails, the cycle will repeat

"bagof" collects all solutions to "can_fire"

```
select_rule(SelectedRule) :-
    bagof(Rule, can_fire(Rule), Candidates),
    resolve(Candidates, SelectedRule).
```

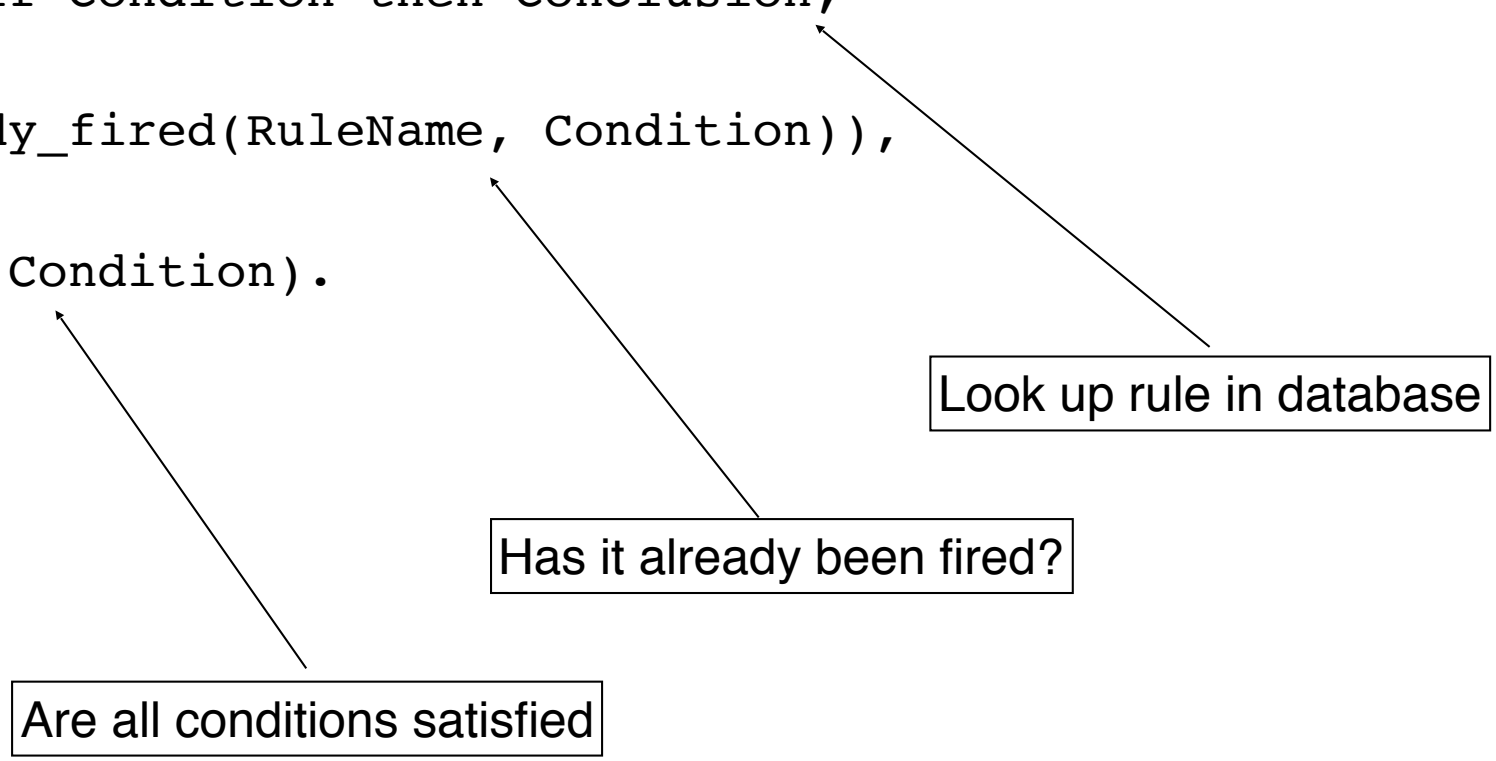
Finding rules that can fire

```
can_fire(RuleName if Condition then Conclusion) :-

    RuleName if Condition then Conclusion,

    not(already_fired(RuleName, Condition)),

    satisfied(Condition).
```



Satisfying Condition

```
satisfied(A and B) :- !,
    wm(A),
    satisfied(B).
satisfied(A) :-
    wm(A).
```

- If pattern is "A and ..." then look for A in working memory and then check rest recursively.
- (A and B) = (x and y and z)

  A = x

  B = y and Z
- If pattern is a single predicate. Look it up.
- Note that "!" prevents a conjunction reaching this clause

Conflict Resolution

```
resolve([], []).
resolve([X|_], X).
```

- Pick the first rule
- Check in case no rules were found

Exercise:

extend resolve to perform:

- specificity ordering
- followed by choosing the largest rule
- followed by choosing the first one.

Firing Rules

```
fire(RuleName if Condition then Conclusion) :- !,
    assert(already_fired(RuleName, Condition)),
    add_to_wm(Conclusion),
    fail.
fire(_).
```

Add a new clause to the database

Add all terms in conclusion to database if not already there

Force backtracking so that a new cycle starts

Succeed if no rule is found so that cycle ends

Add new elements to working memory

```
add_to_wm(A and B) :- !,
    assert_if_not_present(A),
    add_to_wm(B).
add_to_wm(A) :-
    assert_if_not_present(A).
```

For each term in condition, add it to working memory if it is not already there.

```
assert_if_not_present(A) :-
    wm(A), !.
assert_if_not_present(A) :-
    assert(wm(A)).
```

- If term is in working memory, don't do anything
- Otherwise, add new term to working memory.

Starting the cycle

```
run :-
    retractall(already_fired(_, _)),
    exec.
```

- Remove from the database all "already_fired" clauses.
- call "exec" to start the cycle.