Artificial Intelligence I

Dr Mateja Jamnik

Computer Laboratory, Room FC18

Telephone extension 63587

Email: mj201@cl.cam.ac.uk

http://www.cl.cam.ac.uk/users/mj201/

Notes III: problem-solving by planning; knowledge representation and reasoning

Introduction to planning

We now look at how an agent might construct a *plan* enabling it to achieve a goal.

Aims:

- to examine the difference between, on the one hand, problem-solving by search, which we have already addressed, and on the other hand, specialised planning algorithms;
- to look in detail at the basic *partial-order planning algorithm*.

Reading: Russell and Norvig, chapter 11.

Problem solving is different to planning

In search problems we:

- **Represent states:** and a state representation contains *everything* that's relevant about the environment.
- **Represent actions:** by describing a new state obtained from a current state.
- **Represent goals:** all we know is how to test a state either to see if it's a goal, or using a heuristic.
- **A sequence of actions is a 'plan':** but we only consider *sequences of consecutive actions*.

Problem solving is different to planning

Representing a problem such as: 'obtain a copy of the course text book' is hopeless:

- There are far too many possible actions at each step.
- A heuristic can only help you rank states. In particular it does not help you *ignore* useless actions.
- We are forced to start at the initial state, but you have to work out *how* to get the book—that is, go to the library, borrow it from a friend *etc*—*before* you can start to do it.

Planning algorithms work differently

Difference 1:

- planning algorithms use a language, often First-Order Logic (FOL) (or a subset of FOL) to represent states, goals, and actions;
- states and goals are described by sentences;
- actions are described by stating their *preconditions* and their *effects*.

So if you know the goal includes (maybe among other things)

`Have(AI_book)`

and action `Borrow(x)` has an effect `Have(x)` then you know that a plan including

`Borrow(AI_book)`

might be good.

Planning algorithms work differently

Difference 2:

- Planners can add actions at *any relevant point at all*, not just at the end of a sequence starting at the start state.
- This makes sense: I may determine that `Have(Car_keys)` is a good state to be in without worrying about what happens before or after finding them.
- By making an important decision, like requiring `Have(Car_keys)`, early on we may reduce branching and backtracking.
- State descriptions are not complete—`Have(Car_keys)` describes a *class* of states—and this adds flexibility.

Planning algorithms work differently

Difference 3:

It is assumed that most elements of the environment are *independent* of most other elements.

- A goal including several requirements can be attacked with a divide-and-conquer approach.
- Each individual requirement can be fulfilled using a subplan...
- ...and the subplans then combined.

This works provided there is not significant interaction between the subplans.

Running example: gorilla-based mischief

We will use the following simple example problem, which as based on a similar one due to Russell and Norvig.

The intrepid little scamps in the *Cambridge University Roof-Climbing Society* wish to attach an inflatable gorilla to the spire of a famous College. To do this they need to leave home and obtain:

- **An inflatable gorilla:** these can be purchased from all good joke shops.
- **Some rope:** available from a hardware store.
- **A first-aid kit:** also available from a hardware store.

They need to return home after they've finished their shopping.

How do they go about planning their jolly escapade?

Logical inference is different to planning

This problem could certainly be attacked using situation calculus:

Start state:

$$\begin{aligned}\texttt{At}(\texttt{Home}, S_0) &\wedge \neg\texttt{Have}(\texttt{Gorilla}, S_0)\\ &\wedge \neg\texttt{Have}(\texttt{Rope}, S_0)\\ &\wedge \neg\texttt{Have}(\texttt{Kit}, S_0)\end{aligned}$$

Goal:

$$\begin{aligned}\exists s\ \texttt{At}(\texttt{Home}, s) &\wedge \texttt{Have}(\texttt{Gorilla}, s)\\ &\wedge \texttt{Have}(\texttt{Rope}, s)\\ &\wedge \texttt{Have}(\texttt{Kit}, s)\end{aligned}$$

Logical inference is different to planning

Operators:

$$\begin{aligned}&\texttt{Have}(\texttt{Gorilla},\texttt{Result}(a, s)) \iff\\ &\qquad(\texttt{Have}(\texttt{Gorilla}, s) \wedge a \neq \texttt{Drop}(\texttt{Gorilla}))\\ &\qquad\vee (\texttt{At}(\texttt{JokeShop}, s) \wedge a = \texttt{Buy}(\texttt{Gorilla}))\end{aligned}$$

It is possible to use automated logical inference to obtain a *sequence* of actions.

Unfortunately this is highly inefficient.

We need to restrict the *language*, and we need to use a *special-purpose* planning algorithm rather than a general theorem-prover.

The STRIPS language

STRIPS: “Stanford Research Institute Problem Solver” (1970).

States: are *conjunctions* of *ground literals* with *no functions*.

$$\begin{aligned}\texttt{At}(\texttt{Home}) &\wedge \neg\texttt{Have}(\texttt{Gorilla})\\ &\wedge \neg\texttt{Have}(\texttt{Rope})\\ &\wedge \neg\texttt{Have}(\texttt{Kit})\end{aligned}$$

Goals: are *conjunctions* of *literals* where variables are assumed existentially quantified.

$$\texttt{At}(x) \wedge \texttt{Sells}(x, \texttt{Gorilla})$$

A planner finds a sequence of actions that makes the goal true when performed. This is different to a theorem-prover.

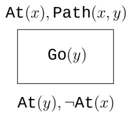
The STRIPS language

STRIPS uses *operators* specifying:

- An *action description*: what the action does.
- A *precondition*: what must be true before the operator can be used. A conjunction of positive literals.
- An *effect*: what is true after the operator has been used. A conjunction of literals.

The STRIPS language

For example:



Op(Action: $\mathtt{Go}(y)$,
Pre: $\mathtt{At}(x) \land \mathtt{Path}(x,y)$
Effect: $\mathtt{At}(y) \land \neg\mathtt{At}(x)$)

All variables are universally quantified.

The space of situations

Standard search algorithms could be used with STRIPS to construct sequences of actions working forward from the start state. This is:

- a *situation space* planner;
- a *progression* planner. It searches from initial state to goal.

A *regression planner* exploits the new language by searching backward from the goal.

This can still be too inefficient.

The space of plans

Alternatively we can search in *plan space*:

- start with an empty plan;
- operate on it to obtain new plans;
- continue until we obtain a plan that solves the problem.

Operations on plans can be:

- adding a step;
- instantiating a variable;
- imposing an ordering that places a step in front of another;
- and so on.

The space of plans

Incomplete plans are called *partial plans*.

Refinement operators add constraints to a partial plan.

All other operators are called *modification operators*.

Representing a plan: partial order planners

When putting on your shoes and socks:

- it *does not matter* whether you deal with your left or right foot first;
- it *does matter* that you place a sock on *before* a shoe, for any given foot.

It makes sense in constructing a plan, *not* to make any *commitment* to which side is done first *if you don't have to*.

Representing a plan: partial order planners

Principle of least commitment: do not commit to any specific choices until you have to. This can be applied both to ordering and to instantiation of variables.

A *partial order planner* allows plans to specify that some steps must come before others but others have no ordering.

A *linearision* of such a plan imposes a specific sequence on the actions therein.

Representing a plan: partial order planners

A plan consists of:

1. A set $\{S_1, S_2, \ldots, S_n\}$ of steps. Each of these is one of the available operators.
2. A set of *ordering constraints*. An ordering constraint $S_i < S_j$ denotes the fact that step S_i must happen before step S_j. $S_i < S_j < S_k$ and so on has the obvious meaning. $S_i < S_j$ does *not* mean that S_i must *immediately* precede S_j.
3. A set of variable bindings $v = x$ where v is a variable and x is either a variable or a constant.
4. A set of *causal links* or *protection intervals* $S_i \xrightarrow{c} S_j$. This denotes the fact that the purpose of S_i is to achieve the precondition c for S_j.

Representing a plan: partial order planners

The *initial plan* has:

- two steps, called `Start` and `Finish`;
- a single ordering constraint `Start` $<$ `Finish`;
- no variable bindings;
- no causal links.

In addition to this:

- the step `Start` has no preconditions, and its effect is the start state for the problem;
- the step `Finish` has no effect, and its precondition is the goal;
- neither `Start` or `Finish` has an associated action.

Solutions to planning problems

A solution to a planning problem is any *complete* and *consistent* partially ordered plan.

Complete: each precondition of each step is *achieved* by another step in the solution.

A precondition c for S is achieved by a step S' if:

1. the precondition is an effect of the step

   $$S' < S \text{ and } c \in \text{Effects}(S')$$

   and;
2. there is no *other* step that can cancel the precondition:

   $$\text{no } S'' \text{ exists where } S' < S'' < S \text{ and } \neg c \in \text{Effects}(S'')$$

Solutions to planning problems

Consistent: no contradictions exist in the binding constraints or in the proposed ordering. That is:

1. for binding constraints, we never have $v = X$ and $v = Y$ for distinct constants X and Y;
2. for the ordering, we never have $S < S'$ and $S' < S$.

An example of partial-order planning

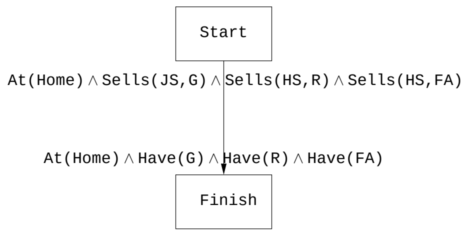
Returning to the roof-climber's shopping expedition.

Here is the basic approach:

- start with only the `Start` and `Finish` steps in the plan;
- at each stage add a new step;
- always add a new step such that a currently non-achieved precondition is achieved;
- backtrack when necessary.

An example of partial-order planning

Here is the initial plan:



Thin arrows denote ordering.

An example of partial-order planning

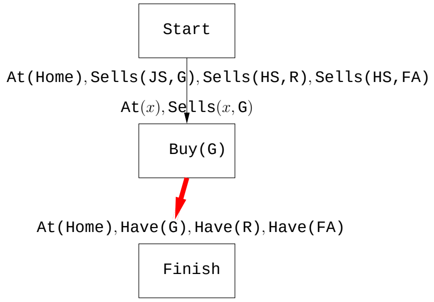
There are two actions available:


$\texttt{At}(x)$
$\texttt{Go}(y)$
$\texttt{At}(y), \neg\texttt{At}(x)$
$\texttt{At}(x), \texttt{Sells}(x, y)$
$\texttt{Buy}(y)$
$\texttt{Have}(y)$

A planner might begin, for example, by adding a `Buy(G)` action in order to achieve the `Have(G)` precondition of `Finish`.

Note: the following order of events is by no means the only one available to a planner. It has been chosen for illustrative purposes.

An example of partial-order planning


Start
$\texttt{At(Home)}, \texttt{Sells(JS,G)}, \texttt{Sells(HS,R)}, \texttt{Sells(HS,FA)}$
$\texttt{At}(x), \texttt{Sells}(x, \texttt{G})$
Buy(G)
$\texttt{At(Home)}, \texttt{Have(G)}, \texttt{Have(R)}, \texttt{Have(FA)}$
Finish

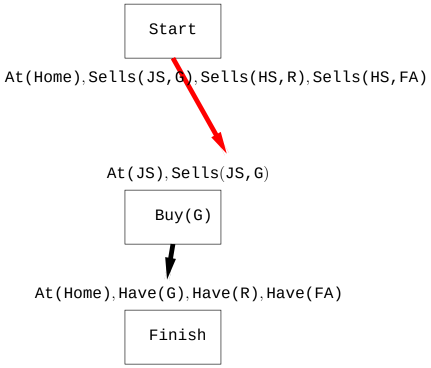
Thick arrows denote causal links.

Here, the new Buy step achieves the `Have(G)` precondition of `Finis`

Thick arrows can be thought of as having a thin arrow underneath.

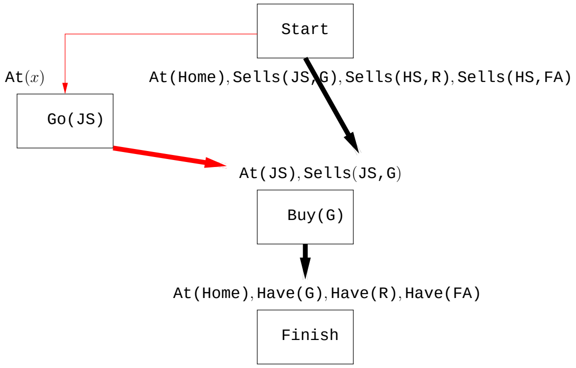
An example of partial-order planning

The planner can now introduce a second causal link from `Start` to achieve the $\texttt{Sells}(x, \texttt{G})$ precondition of `Buy(G)`.


Start
$\texttt{At(Home)}, \texttt{Sells(JS,G)}, \texttt{Sells(HS,R)}, \texttt{Sells(HS,FA)}$
$\texttt{At(JS)}, \texttt{Sells}(\texttt{JS}, \texttt{G})$
Buy(G)
$\texttt{At(Home)}, \texttt{Have(G)}, \texttt{Have(R)}, \texttt{Have(FA)}$
Finish

An example of partial-order planning

The planner's next obvious move is to introduce a Go step to achieve the `At(HS)` precondition of `Buy(G)`.

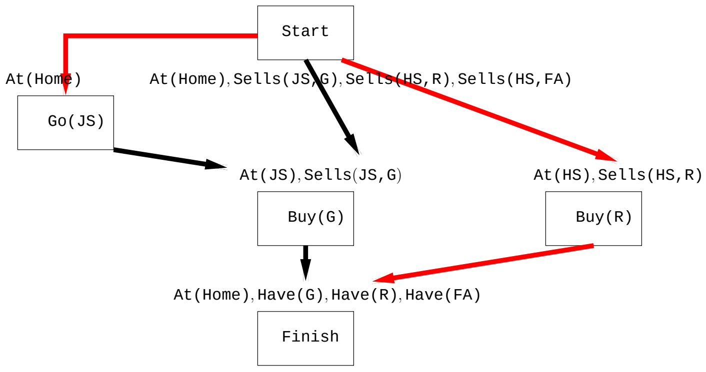

Start
$\texttt{At}(x)$
$\texttt{At(Home)}, \texttt{Sells(JS,G)}, \texttt{Sells(HS,R)}, \texttt{Sells(HS,FA)}$
Go(JS)
$\texttt{At(JS)}, \texttt{Sells}(\texttt{JS}, \texttt{G})$
Buy(G)
$\texttt{At(Home)}, \texttt{Have(G)}, \texttt{Have(R)}, \texttt{Have(FA)}$
Finish

An example of partial-order planning

Initially the planner can continue quite easily in this manner:

- Add a causal link from Start to Go(JS) to achieve the At(x) precondition.
- Add the step Buy(R) with an associated causal link to the Have(R) precondition of Finish.
- Add a causal link from Start to Buy(R) to achieve the Sells(HS,R) precondition.

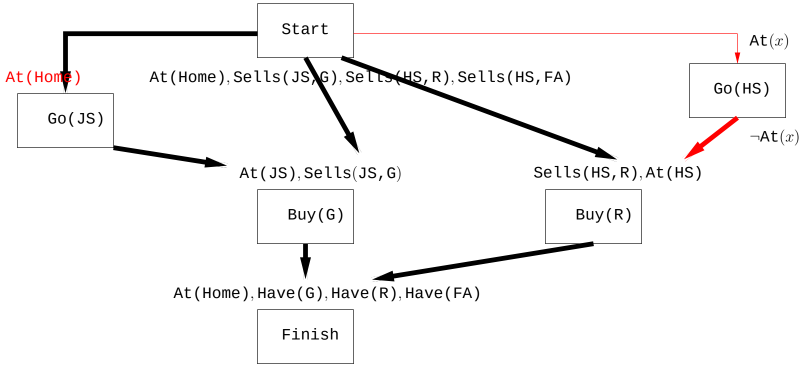
An example of partial-order planning



At this point it starts to get tricky...

The At(HS) precondition in Buy(R) is not achieved.

An example of partial-order planning



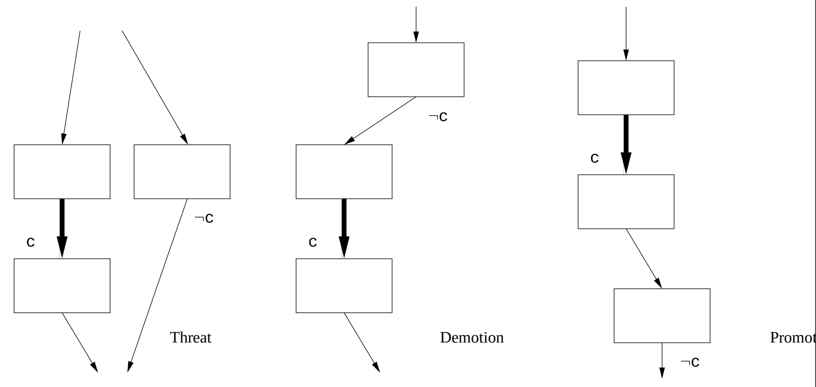
The At(HS) precondition is easy to achieve.

But if we introduce a causal link from Start *to* Go(HS) *then we risk invalidating the precondition for* Go(JS).

An example of partial-order planning

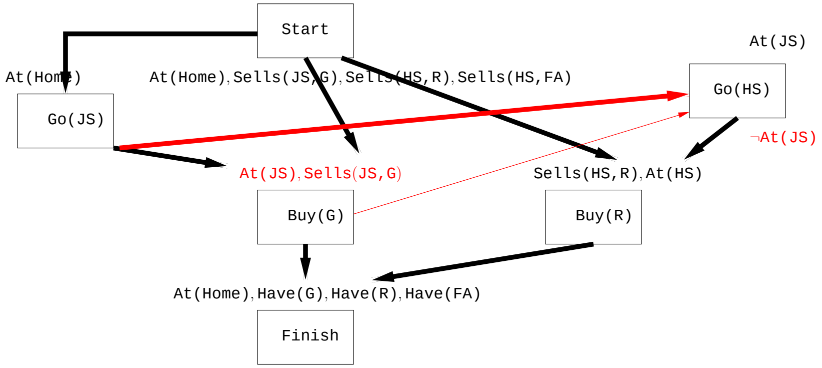
A step that might invalidate (sometimes the word *clobber* is employed) a previously achieved precondition is called a *threat*.

A planner can try to fix a threat by introducing an ordering constraint.



An example of partial-order planning

The planner could backtrack and try to achieve the At(x) precondition using the existing `Go(JS)` step.



This involves a threat, but one that can be fixed using promotion.

The algorithm

```
plan partial_order_plan(start,finish,ops)
{
  plan=empty_plan(start,finish);

  while(true)
  {
    if (solution(plan))
      return plan;
    else
    {
      (step,pre)=get_subgoal(plan);
      choose_op(plan,ops,step,pre);
      resolve_threats(plan);
    }
  }
}
```

The algorithm

```
(step,pre) get_subgoal(plan)
{
  pick some step from steps in plan for which
  a precondition pre is not yet achieved;

  return (step,pre);
}
```

The algorithm

```
choose_op(plan,ops,step,pre)
{
  choose S from ops or current steps in plan
  having effect pre;

  if (no S exists)
    fail;
  include a causal link from S to step in the plan;
  include S < step in the plan;
  if(S doesn’t yet appear in the plan)
  {
    add S;
    add Start < S < Finish;
  }
}
```

The algorithm

```
resolve_threats(plan)
{
  for (all steps S threatening some causal link from
  S’ to S’’)
  {
    choose
      1. add S < S’ to the plan (promotion)
      2. add S’’ < S’ to the plan (demotion)

    if (the plan is not consistent)
      fail;
  }
}
```

Possible threats

If at any stage an effect $\neg$`At`(x) appears, is it a threat to `At(JS)`?

Such an occurrence is called a *possible threat* and an algorithm can be made to deal with it in three different ways:

1. use an equality constraint to resolve immediately;
2. use an inequality constraint to resolve immediately;
3. leave the choice of x’s value until later.

Introduction to knowledge representation and reasoning

We now look briefly at how knowledge about the world might be represented and reasoned with.

Aims:

- To introduce *semantic networks* and *frames* for knowledge representation.
- To see how *inheritance* can be applied as a reasoning method.
- To look at the use of *rules* for knowledge representation, along with *forward chaining* and *backward chaining* for reasoning.

Reading: *The Essence of Artificial Intelligence*, Alison Cawsey. Prentice Hall, 1998.

Knowledge representation

The “manipulation of knowledge” seems to be at the heart of what we as intelligent beings do.

To try to model this process in an agent we:

- *represent* knowledge using *symbol structures*, and;
- perform *formalised* versions of reasoning.

This means that we need carefully specified *languages* for the representation of knowledge.

Requirements for a knowledge representation language

First, we need **representational adequacy**.

Can I represent the pieces of knowledge I need to?

Propositional logic might well fail this test, although *predicate logic* seems better and is indeed a standard tool.

Requirements for a knowledge representation language

Or more subtly:

Can I represent the pieces of knowledge I need to in such a way that reasoning can be automated?

English is excellent and highly expressive in representing knowledge:

"Ophelia believes that all sensible people dislike eating pies"

However automating reasoning based on English language representations is just about impossible at present.

How would we write a program that takes this statement and when told *"Neddy is really jolly sensible"* and *"Neddy is a funny sort of person"* infers that *"Ophelia believes Neddy dislikes eating pies"*?

Requirements for a knowledge representation language

On the other hand:

$$\texttt{person}(\texttt{neddy})$$

$$\texttt{sensible}(\texttt{neddy})$$

$$\forall x\ \texttt{sensible}(x) \land \texttt{person}(x) \rightarrow (\forall y\ \texttt{pie}(y) \rightarrow \texttt{dislikes}(x, y))$$

is something for which reasoning can be automated.

Syntax and semantics

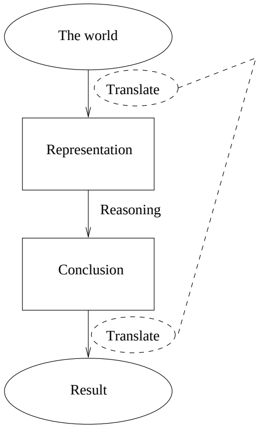
In addition to needing an expressive language, the language needs to be clearly defined:

- **Syntax:** defining when a statement in the language is well-formed.
- **Semantics**: specifying what a statement in the language *means*.

English is again not good here from the point of view of automation. Logic is again preferable.

If possible, we also want the representation to be *natural* in the sense that it is reasonably easy to understand and deal with.

Syntax and semantics



To translate confidently we need
well-defined syntax and semantics

Does

`dislikes(neddy, pie)`

mean Neddy dislikes all pies, or that he's just taken a dislike to an individual pie?

Inferential adequacy and inferential efficiency

We also need to know that we can infer the things of interest:

- It is not always possible, and it's certainly not desirable, to store all knowledge as explicit facts.

  Knowing that *"all dogs smell bad"* should allow us to infer that *"fido smells bad" etc*. We don't want to store a piece of knowledge for every possible dog.
- However, more complex inferences are likely to take longer.

So as usual, there is a trade-off.

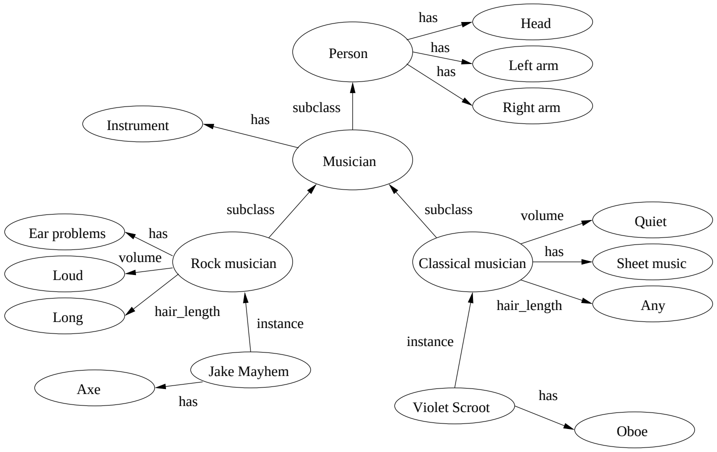
Frames and semantic networks

Frames and semantic networks represent knowledge in the form of *classes of objects* and *relationships between them*:

- the *subclass* and *instance* relationships are emphasised;
- we form *class hierarchies* in which *inheritance* is supported and provides the main *inference* mechanism;
- as a result inference is quite limited;
- we need to be extremely careful about *semantics*.

The only major difference between the two ideas is *notational*.

Example of a semantic network



Frames

Frames once again support inheritance through the subclass relationship.

Rock musician

subclass: Musician
has: ear problems
hairlength: long
volume: loud

Musician

subclass: Person
has: instrument

has, hairlength, volume *etc* are “slots”.

long, loud, instrument *etc* are “slot values”.

These are a direct predecessor of object-oriented programming languages.

Defaults

Both approaches to knowledge representation are able to incorporate *defaults*:

Rock musician

subclass: Musician
has: ear problems
* hairlength: long
* volume: loud

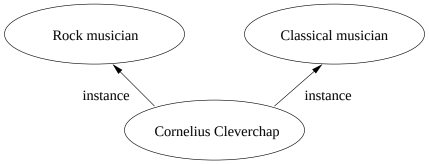
Dementia Evilperson

subclass: Rock musician
hairlength: short
image: gothic

Starred slots are *typical* values associated with subclasses and instances, but can be overridden.

Multiple inheritance

Both approaches can incorporate *multiple inheritance*, at a cost:



- what is `hairlength` for `Cornelius` if we’re trying to use inheritance to establish it?
- this can be overcome initially by specifying which class is inherited from in preference when there’s a conflict;
- but the problem is still not entirely solved—what if we want to prefer inheritance of some things from one class, but inheritance of others from a different one?

Other issues

- Slots and slot values can themselves be frames. For example `Dementia` may have an instrument slot with the value `Electric` which itself may have properties described in a frame.
- Slots can have *specified attributes*. For example, we might specify that `instrument` can have multiple values, that each value can only be an instance of `Instrument` that each value has a slot called `owned_by` and so on.
- Slots may contain arbitrary pieces of program. This is known as *procedural attachment*. The fragment might be executed to return the slot’s value, or update the values in other slots *etc*.

Rule-based systems

A rule-based system requires three things:

1. A set of `if-then` rules. These denote specific pieces of knowledge about the world.

   They should be interpreted similarly to logical implication, rather than the programming construct. In particular a collection of such rules doesn't necessarily imply a sequence.

   Such rules denote *what to do* or *what can be inferred* under given circumstances.
2. A collection of *facts* denoting what the system regards as currently true about the world.
3. An interpreter able to apply the current rules in the light of the current facts.

Forward chaining

The first of two basic kinds of interpreter begins with established facts and then applies rules to them.

This is a *data-driven* process. It is appropriate if we know the *initial facts* but not the required conclusion.

Example: XCON—used for configuring VAX computers.

In addition:

- we maintain a *working memory*, typically of what has been inferred so far;
- rules are often *condition-action rules*, where the right-hand side specifies an action such as adding or removing something from working memory, printing a message *etc*;
- in some cases actions might be entire program fragments.

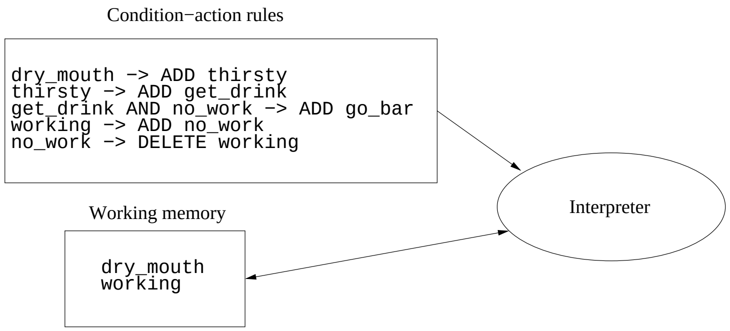
Forward chaining

The basic algorithm is:

1. find all the rules that can fire, based on the current working memory;
2. select a rule to fire. This requires a *conflict resolution strategy*;
3. carry out the action specified, possibly updating the working memory.

Repeat this process until either no rules can be used or a "halt" appears in the working memory.

Example



```
dry_mouth -> ADD thirsty
thirsty -> ADD get_drink
get_drink AND no_work -> ADD go_bar
working -> ADD no_work
no_work -> DELETE working
```

```
dry_mouth
working
```

Example

Progress is as follows:

1. The rule

   `dry_mouth` → ADD `thirsty`

   fires adding `thirsty` to working memory.
2. The rule

   `thirsty` → ADD `get_drink`

   fires adding `get_drink` to working memory.
3. The rule

   `working` → ADD `no_work`

   fires adding `no_work` to working memory.
4. The rule

   `get_drink` AND `no_work` → ADD `go_bar`

   fires, and we establish that it's time to go to the bar.

Conflict resolution

Clearly, in any more realistic system we expect to have to deal with a scenario where two or more rules can be fired at any one time:

- which rule we choose can clearly affect the outcome;
- we might also want to attempt to avoid inferring an abundance of useless information.

We therefore need a means of resolving such conflicts.

Conflict resolution

Common *conflict resolution* strategies are:

- prefer rules involving more recently added facts;
- prefer rules that are *more specific*. For example

  `patient_coughing` → ADD `lung_problem`

  is more general than

  `patient_coughing` AND `patient_smoker` → ADD `lung_cancer`.

  This allows us to define exceptions to general rules;
- allow the designer of the rules to specify priorities;
- fire all rules simultaneously—this essentially involves following all chains of inference at once.

Reason maintenance

Some systems will allow information to be removed from the working memory if it is no longer *justified*.

For example, we might find that

`patient_coughing`

and

`patient_smoker`

are in working memory, and hence fire

`patient_coughing` AND `patient_smoker` → ADD `lung_cancer`

but later infer something that causes `patient_coughing` to be withdrawn from working memory.

The justification for `lung_cancer` has been removed, and so it should perhaps be removed also.

Pattern matching

In general rules may be expressed in a slightly more flexible form involving *variables* which can work in conjunction with *pattern matching*.

For example the rule

$$\texttt{coughs}(X)\ \text{AND}\ \texttt{smoker}(X) \rightarrow \text{ADD}\ \texttt{lung\_cancer}(X)$$

contains the variable X.

If the working memory contains $\texttt{coughs}(\texttt{neddy})$ and $\texttt{smoker}(\texttt{neddy})$ then

$$X = \texttt{neddy}$$

provides a match and

$$\texttt{lung\_cancer}(\texttt{neddy})$$

is added to the working memory.

Backward chaining

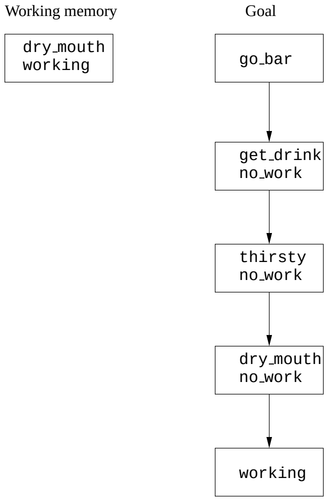
The second basic kind of interpreter begins with a *goal* and finds a rule that would achieve it.

It then works backwards, trying to achieve the resulting earlier goals in the succession of inferences.

Example: MYCIN—medical diagnosis with a small number of conditions.

This is a *goal-driven* process. If you want to *test a hypothesis* or you have some idea of a likely conclusion it can be more efficient than forward chaining.

Example



To establish `go_bar` we have to establish `get_drink` and `no_work`. These are the new goals.

Try first to establish `get_drink`. This can be done by establishing `thirsty`.

`thirsty` can be established by establishing `dry_mouth`. This is in the working memory so we're done.

Finally, we can establish `no_work` by establishing `working`. This is in the working memory so the process has finished.

Example with backtracking

If at some point more than one rule has the required conclusion then we can *backtrack*.

Example: *Prolog* backtracks, and incorporates pattern matching. It orders attempts according to the order in which rules appear in the program.

Example: having added

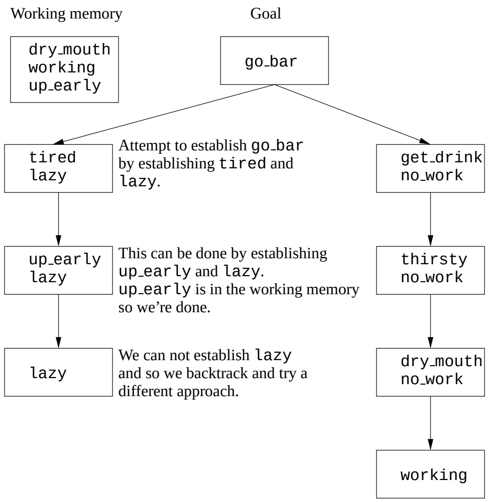
$$\texttt{up\_early} \rightarrow \text{ADD}\ \texttt{tired}$$

and

$$\texttt{tired}\ \text{AND}\ \texttt{lazy} \rightarrow \text{ADD}\ \texttt{go\_bar}$$

to the rules, and `up_early` to the working memory:

Example with backtracking



Attempt to establish `go_bar` by establishing `tired` and `lazy`.

This can be done by establishing `up_early` and `lazy`. `up_early` is in the working memory so we're done.

We can not establish `lazy` and so we backtrack and try a different approach.

Process proceeds as before