14.

Actions

Situation calculus

The <u>situation calculus</u> is a dialect of FOL for representing dynamically changing worlds in which all changes are the result of named actions.

There are two distinguished sorts of terms:

- <u>actions</u>, such as
  - $\mathrm{put}(x,y)$ put object x on top of object y
  - $\mathrm{walk}(loc)$ walk to location loc
  - $\mathrm{pickup}(r,x)$ robot r picks up object x
- <u>situations</u>, denoting possible world histories. A distinguished constant S_0 and function symbol do are used
  - S_0 the initial situation, before any actions have been performed
  - $do(a,s)$ the situation that results from doing action a in situation s

for example: $do(\mathrm{put}(A,B),do(\mathrm{put}(B,C),S_0))$ the situation that results from putting A on B after putting B on C in the initial situation

Fluents

Predicates or functions whose values may vary from situation to situation are called <u>fluents</u>.

These are written using predicate or function symbols whose last argument is a situation

for example: $\mathrm{Holding}(r, x, s)$: robot r is holding object x in situation s

can have: $\neg\mathrm{Holding}(r, x, s) \wedge \mathrm{Holding}(r, x, do(\mathrm{pickup}(r,x),s))$

the robot is not holding the object x in situation s, but is holding it in the situation that results from picking it up

Note: there is no distinguished "current" situation. A sentence can talk about many different situations, past, present, or future.

A distinguished predicate symbol $Poss(a,s)$ is used to state that a may be performed in situation s

for example: $Poss(\mathrm{pickup}(r,x), S_0)$ it is possible for the robot r to pickup object x in the initial situation

This is the entire language.

Preconditions and effects

It is necessary to include in a KB not only facts about the initial situation, but also about world dynamics: what the actions do.

Actions typically have <u>preconditions</u>: what needs to be true for the action to be performed

- $Poss(\mathrm{pickup}(r,x), s) \equiv \forall z.\ \neg\mathrm{Holding}(r,z,s) \wedge \neg\mathrm{Heavy}(x) \wedge \mathrm{NextTo}(r,x,s)$

  a robot can pickup an object iff it is not holding anything, the object is not too heavy, and the robot is next to the object

  Note: free variables assumed to be universally quantified
- $Poss(\mathrm{repair}(r,x), s) \equiv \mathrm{HasGlue}(r,s) \wedge \mathrm{Broken}(x,s)$

  it is possible to repair an object iff the object is broken and the robot has glue

Actions typically have <u>effects</u>: the fluents that change as the result of performing the action

- $\mathrm{Fragile}(x) \supset \mathrm{Broken}(x, do(\mathrm{drop}(r,x),s))$

  dropping a fragile object causes it to break
- $\neg\mathrm{Broken}(x, do(\mathrm{repair}(r,x),s))$

  repairing an object causes it to be unbroken

The frame problem

To really know how the world works, it is also necessary to know what fluents are *unaffected* by performing an action.

- $\text{Colour}(x,c,s) \supset \text{Colour}(x, c, do(\text{drop}(r,x),s))$

  dropping an object does not change its colour

- $\neg\text{Broken}(x,s) \wedge [\, x{\neq}y \vee \neg\text{Fragile}(x)\,] \supset \neg\text{Broken}(x, do(\text{drop}(r,y),s)$

  not breaking things

These are sometimes called frame axioms.

Problem: need to know a vast number of such axioms. (Few actions affect the value of a given fluent; most leave it invariant.)

an object's colour is unaffected by picking things up, opening a door, using the phone, turning on a light, electing a new Prime Minister of Canada, *etc.*

The frame problem:

- in building KB, need to think of these $\sim 2 \times A \times F$ facts about what does not change
- the system needs to reason efficiently with them

What counts as a solution?

- Suppose the person responsible for building a KB has written down *all* the effect axioms

  for each fluent F and action A that can cause the truth value of F to change, an axiom of the form $[R(s) \supset \pm F(do(A,s))]$, where $R(s)$ is some condition on s

- We want a systematic procedure for generating all the frame axioms from these effect axioms
- If possible, we also want a *parsimonious* representation for them (since in their simplest form, there are too many)

Why do we want such a solution?

- frame axioms are necessary to reason about actions and are not entailed by the other axioms
- convenience for the KB builder
- for theorizing about actions

– modularity: only add effect axioms

– accuracy: no inadvertent omissions

The projection task

What can we do with the situation calculus?

We will see later that it can be used for planning.

A simpler job we can handle directly is called the projection task.

Given a sequence of actions, determine what would be true in the situation that results from performing that sequence.

This can be formalized as follows:

Suppose that $R(s)$ is a formula with a free situation variable s.

To find out if $R(s)$ would be true after performing $\langle a_1,...,a_n\rangle$ in the initial situation, we determine whether or not

$$KB \models R(do(a_n,do(a_{n-1},...,do(a_1,S_0)...)))$$

For example, using the effect and frame axioms from before, it follows that $\neg\text{Broken}(\text{B},s)$ would hold after doing the sequence

$$\langle \text{pickup(A), pickup(B), drop(B), repair(B), drop(A)} \rangle$$

The legality task

The projection task above asks if a condition would hold after performing a sequence of actions, but not whether that sequence can in fact be properly executed.

We call a situation legal if it is the initial situation or the result of performing an action whose preconditions are satisfied starting in a legal situation.

The legality task is the task of determining whether a sequence of actions leads to a legal situation.

This can be formalized as follows:

To find out if the sequence $\langle a_1,...,a_n\rangle$ can be legally performed in the initial situation, we determine whether or not

$$KB \models Poss(a_i, do(a_{i-1},...,do(a_1,S_0)...))$$

for every i such that $1 \le i \le n$.

Limitations of the situation calculus

This version of the situation calculus has a number of limitations:

- no time: cannot talk about how long actions take, or when they occur
- only known actions: no hidden exogenous actions, no unnamed events
- no concurrency: cannot talk about doing two actions at once
- only discrete situations: no continuous actions, like pushing an object from A to B.
- only hypotheticals: cannot say that an action has occurred or will occur
- only primitive actions: no actions made up of other parts, like conditionals or iterations

We will deal with the last of these below.

First we consider a simple solution to the frame problem ...

Normal form for effect axioms

Suppose there are two positive effect axioms for the fluent *Broken:*

$\mathrm{Fragile}(x) \supset \mathrm{Broken}(x, do(\mathrm{drop}(r,x),s))$

$\mathrm{NextTo}(b,x,s) \supset \mathrm{Broken}(x, do(\mathrm{explode}(b),s))$

These can be rewritten as

$\exists r\, \{a=\mathrm{drop}(r,x) \wedge \mathrm{Fragile}(x)\} \vee \exists b\, \{a=\mathrm{explode}(b) \wedge \mathrm{NextTo}(b,x,s)\}$
$\supset \mathrm{Broken}(x, do(a,s))$

Similarly, consider the negative effect axiom:

$\neg\mathrm{Broken}(x, do(\mathrm{repair}(r,x),s))$

which can be rewritten as

$\exists r\, \{a=\mathrm{repair}(r,x)\} \supset \neg\mathrm{Broken}(x, do(a,s))$

In general, for any fluent F, we can rewrite all the effect axioms as as two formulas of the form

$\mathrm{P}_F(\boldsymbol{x}, a, s) \supset F(\boldsymbol{x}, do(a,s))$ (1)

$\mathrm{N}_F(\boldsymbol{x}, a, s) \supset \neg F(\boldsymbol{x}, do(a,s))$ (2)

where $\mathrm{P}_F(\boldsymbol{x}, a, s)$ and $\mathrm{N}_F(\boldsymbol{x}, a, s)$ are formulas whose free variables are among the x_i, a, and s.

Explanation closure

Now make a completeness assumption regarding these effect axioms:

assume that (1) and (2) characterize *all* the conditions under which an action a changes the value of fluent F.

This can be formalized by <u>explanation closure axioms</u>:

$\neg F(\boldsymbol{x}, s) \wedge F(\boldsymbol{x}, do(a,s)) \supset \mathrm{P}_F(\boldsymbol{x}, a, s)$ (3)

if F was false and was made true by doing action a then condition P_F must have been true

$F(\boldsymbol{x}, s) \wedge \neg F(\boldsymbol{x}, do(a,s)) \supset \mathrm{N}_F(\boldsymbol{x}, a, s)$ (4)

if F was true and was made false by doing action a then condition N_F must have been true

These explanation closure axioms are in fact disguised versions of frame axioms!

$\neg F(\boldsymbol{x}, s) \wedge \neg\mathrm{P}_F(\boldsymbol{x}, a, s) \supset \neg F(\boldsymbol{x}, do(a,s))$

$F(\boldsymbol{x}, s) \wedge \neg\mathrm{N}_F(\boldsymbol{x}, a, s) \supset F(\boldsymbol{x}, do(a,s))$

Successor state axioms

Further assume that our KB entails the following

- integrity of the effect axioms: $\neg\exists\, \boldsymbol{x}, a, s.\ \mathrm{P}_F(\boldsymbol{x}, a, s) \wedge \mathrm{N}_F(\boldsymbol{x}, a, s)$
- unique names for actions:

  $A(x_1,\ldots,x_n) = A(y_1,\ldots,y_n) \supset (x_1=y_1) \wedge \ldots \wedge (x_n=y_n)$

  $A(x_1,\ldots,x_n) \neq B(y_1,\ldots,y_m)$ where A and B are distinct

Then it can be shown that KB entails that (1), (2), (3), and (4) together are logically equivalent to

$$F(\boldsymbol{x}, do(a,s)) \equiv \mathrm{P}_F(\boldsymbol{x}, a, s) \vee (F(\boldsymbol{x}, s) \wedge \neg\mathrm{N}_F(\boldsymbol{x}, a,s))$$

This is called the <u>successor state axiom</u> for F.

For example, the successor state axiom for the *Broken* fluent is:

$\mathrm{Broken}(x, do(a,s)) \equiv$
$\exists r\, \{a=\mathrm{drop}(r,x) \wedge \mathrm{Fragile}(x)\}$
$\vee\ \exists b\, \{a=\mathrm{explode}(b) \wedge \mathrm{NextTo}(b,x,s)\}$
$\vee\ \mathrm{Broken}(x, s) \wedge \neg\exists r\, \{a=\mathrm{repair}(r,x)\}$

An object x is broken after doing action a
iff
a is a dropping action and x is fragile,
or a is a bomb exploding where x is next to the bomb,
or x was already broken and a is not the action of repairing it

Note universal quantification: for *any* action a ...

A simple solution to the frame problem

This simple solution to the frame problem (due to Ray Reiter) yields the following axioms:

- one successor state axiom per fluent
- one precondition axiom per action
- unique name axioms for actions

Moreover, we do not get fewer axioms at the expense of prohibitively long ones

the length of a successor state axioms is roughly proportional to the number of actions which affect the truth value of the fluent

The conciseness and perspicuity of the solution relies on

- quantification over actions
- the assumption that relatively few actions affect each fluent
- the completeness assumption (for effects)

Moreover, the solution depends on the fact that actions always have deterministic effects.

Limitation: primitive actions

As yet we have no way of handling in the situation calculus complex actions made up of other actions such as

- conditionals: If the car is in the driveway then drive else walk
- iterations: while there is a block on the table, remove one
- nondeterministic choice: pickup up some block and put it on the floor

and others

Would like to *define* such actions in terms of the primitive actions, and inherit their solution to the frame problem

Need a compositional treatment of the frame problem for complex actions

Results in a novel programming language for discrete event simulation and high-level robot control

The Do formula

For each complex action A, it is possible to define a formula of the situation calculus, $Do(A, s, s')$, that says that action A when started in situation s may legally terminate in situation s'.

Primitive actions: $Do(A, s, s') \;=\; Poss(A,s) \wedge s'=do(A,s)$

Sequence: $Do([A;B], s, s') \;=\; \exists s''.\, Do(A, s, s'') \wedge Do(B, s'', s')$

Conditionals: $Do([\textbf{if}\ \phi\ \textbf{then}\ A\ \textbf{else}\ B], s, s') \;=$
$\phi(s) \wedge Do(A, s, s') \;\vee\; \neg\phi(s) \wedge Do(B, s, s')$

Nondeterministic branch: $Do([A \mid B], s, s') \;=\; Do(A, s, s') \vee Do(B, s, s')$

Nondeterministic choice: $Do([\pi x.\, A], s, s') \;=\; \exists x.\, Do(A, s, s')$

etc.

Note: programming language constructs with a purely logical situation calculus interpretation

GOLOG

GOLOG (Algol in logic) is a programming language that generalizes conventional imperative programming languages

- the usual imperative constructs + concurrency, nondeterminism, more...
- bottoms out *not* on operations on internal states (assignment statements, pointer updates) but on *primitive actions* in the world (*e.g.* pickup a block)
- what the primitive actions do is user-specified by precondition and successor state axioms

What does it mean to "execute" a GOLOG program?

- find a sequence of primitive actions such that performing them starting in some initial situation s would lead to a situation s' where the formula $Do(A, s, s')$ holds
- give the sequence of actions to a robot for actual execution in the world

Note: to find such a sequence, it will be necessary to reason about the primitive actions

$A\ ;\ \textbf{if}\ \text{Holding}(x)\ \textbf{then}\ B\ \textbf{else}\ C$ to decide between B and C we need to determine if the fluent $Holding$ would be true after doing A

GOLOG example

Primitive actions: pickup(x), putonfloor(x), putontable(x)

Fluents: Holding(x,s), OnTable(x,s), OnFloor(x,s)

Action preconditions: $Poss(\text{pickup}(x), s) \equiv \forall z.\neg\text{Holding}(z, s)$
$Poss(\text{putonfloor}(x), s) \equiv \text{Holding}(x, s)$
$Poss(\text{putontable}(x), s) \equiv \text{Holding}(x, s)$

Successor state axioms:

$\text{Holding}(x, do(a,s)) \equiv a=\text{pickup}(x) \lor \text{Holding}(x,s) \land a\neq\text{putontable}(x) \land a\neq\text{putonfloor}(x)$

$\text{OnTable}(x, do(a,s)) \equiv a=\text{putontable}(x) \lor \text{OnTable}(x,s) \land a\neq\text{pickup}(x)$

$\text{OnFloor}(x, do(a,s)) \equiv a=\text{putonfloor}(x) \lor \text{OnFloor}(x,s) \land a\neq\text{pickup}(x)$

Initial situation: $\forall x.\neg\text{Holding}(x, S_0)$
$\text{OnTable}(x, S_0) \equiv x=\text{A} \lor x=\text{B}$

Complex actions:

proc ClearTable : ***while*** $\exists b.\text{OnTable}(b)$ ***do*** πb [OnTable(b)? ; RemoveBlock(b)]

proc RemoveBlock(x) : pickup(x) **;** putonfloor(x)

Running GOLOG

To find a sequence of actions constituting a legal execution of a GOLOG program, we can use Resolution with answer extraction.

For the above example, we have

$KB \models \exists s.\, Do(\text{ClearTable}, S_0, s)$

The result of this evaluation yields

$s = do(\text{putonfloor(B)}, do(\text{pickup(B)}, do(\text{putonfloor(A)}, do(\text{pickup(A)},S_0))))$

and so a correct sequence is

$\langle$ pickup(A), putonfloor(A), pickup(B), putonfloor(B)$\rangle$

When what is known about the actions and initial state can be expressed as Horn clauses, the evaluation can be done in Prolog.

The GOLOG interpreter in Prolog has clauses like

```
do(A,S1,do(A,S1)) :- prim_action(A), poss(A,S1).
do(seq(A,B),S1,S2) :- do(A,S1,S3), do(B,S3,S2).
```

This provides a convenient way of controlling a robot at a high level.

15. Planning

Planning

So far, in looking at actions, we have considered how an agent could figure out what to do given a high-level program or complex action to execute.

Now, we consider a related but more general reasoning problem: figure out what to do to make an arbitrary condition true. This is called planning.

- the condition to be achieved is called the goal
- the sequence of actions that will make the goal true is called the plan

Plans can be at differing levels of detail, depending on how we formalize the actions involved

- “do errands” vs. “get in car at 1:32 PM, put key in ignition, turn key clockwise, change gears,…”

In practice, planning involves anticipating what the world will be like, but also observing the world and replanning as necessary...

Using the situation calculus

The situation calculus can be used to represent what is known about the current state of the world and the available actions.

The planning problem can then be formulated as follows:

> Given a formula $Goal(s)$, find a sequence of actions $\boldsymbol{a}$ such that
>
> $$KB \models Goal(do(\boldsymbol{a}, S_0)) \wedge Legal(do(\boldsymbol{a}, S_0))$$
>
> where $do(\langle a_1,...,a_n\rangle, S_0)$ is an abbreviation for
>
> $$do(a_n, do(a_{n-1}, ..., do(a_2, do(a_1, S_0)) ...))$$
>
> and where $Legal(\langle a_1,...,a_n\rangle, S_0)$ is an abbreviation for
>
> $$Poss(a_1, S_0) \wedge Poss(a_2, do(a_1, S_0)) \wedge \ ... \wedge Poss(a_n, do(\langle a_1,...,a_{n-1}\rangle, S_0))$$

So: given a goal formula, we want a sequence of actions such that

- the goal formula holds in the situation that results from executing the actions, and
- it is possible to execute each action in the appropriate situation

Planning by answer extraction

Having formulated planning in this way, we can use Resolution with answer extraction to find a sequence of actions:

$$KB \models \exists s.\ Goal(s) \wedge Legal(s)$$

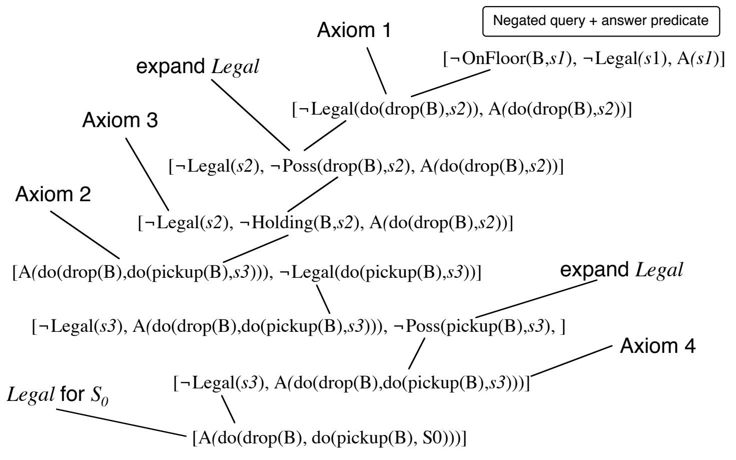
We can see how this will work using a simplified version of a previous example:

An object is on the table that we would like to have on the floor. Dropping it will put it on the floor, and we can drop it, provided we are holding it. To hold it, we need to pick it up, and we can always do so.

- Effects: $\text{OnFloor}(x, do(\text{drop}(x),s))$
  $\text{Holding}(x, do(\text{pickup}(x),s))$
  Note: ignoring frame problem
- Preconds: $\text{Holding}(x, s) \supset Poss(\text{drop}(x), s)$
  $Poss(\text{pickup}(x), s)$
- Initial state: $\text{OnTable}(\text{B}, S_0)$
- The goal: $\text{OnFloor}(\text{B}, s)$

KB

Deriving a plan


Negated query + answer predicate
Axiom 1
[¬OnFloor(B,s1), ¬Legal(s1), A(s1)]
expand Legal
[¬Legal(do(drop(B),s2)), A(do(drop(B),s2))]
Axiom 3
[¬Legal(s2), ¬Poss(drop(B),s2), A(do(drop(B),s2))]
Axiom 2
[¬Legal(s2), ¬Holding(B,s2), A(do(drop(B),s2))]
[A(do(drop(B),do(pickup(B),s3))), ¬Legal(do(pickup(B),s3))]
expand Legal
[¬Legal(s3), A(do(drop(B),do(pickup(B),s3))), ¬Poss(pickup(B),s3),]
Axiom 4
[¬Legal(s3), A(do(drop(B),do(pickup(B),s3)))]
Legal for S0
[A(do(drop(B), do(pickup(B), S0)))]

Here is the plan: in the initial situation, pickup block B, and in the resulting situation, drop B.

Using Prolog

Because all the required facts here can be expressed as Horn clauses, we can use Prolog directly to synthesize a plan:

```
onfloor(X,do(drop(X),S)).
holding(X,do(pickup(X),S)).
poss(drop(X),S) :- holding(X,S).
poss(pickup(X),S).
ontable(b,s0).
legal(s0).
legal(do(A,S)) :- poss(A,S), legal(S).
```

With the Prolog goal `?- onfloor(b,S), legal(S).`
we get the solution `S = do(drop(b),do(pickup(b),s0))`

But planning problems are rarely this easy!

Full Resolution theorem-proving can be problematic for a complex set of axioms dealing with actions and situations explicitly...

The STRIPS representation

STRIPS is an alternative representation to the pure situation calculus for planning.

from work on a robot called Shaky at SRI International in the 60's.

In STRIPS, we do not represent histories of the world, as in the situation calculus.

Instead, we deal with a single world <u>state</u> at a time, represented by a database of ground atomic wffs (e.g., $\mathrm{In}(\mathrm{robot},\mathrm{room}_1)$)

This is like the database of facts used in procedural representations and the working memory of production systems

Similarly, we do not represent actions as part of the world model (cannot reason about them directly), as in the situation calculus.

Instead, actions are represented by <u>operators</u> that syntactically transform world models

An operator takes a DB and transforms it to a new DB

STRIPS operators

Operators have pre- and post-conditions

- precondition = formulas that need to be true at start
- "delete list" = formulas to be removed from DB
- "add list" = formulas to be added to DB

Example: $\mathrm{PushThru}(o,d,r_1,r_2)$

"the robot pushes object o through door d from room r_1 to room r_2"

- precondition: $\mathrm{InRoom}(\mathrm{robot},r_1)$, $\mathrm{InRoom}(o,r_1)$, $\mathrm{Connects}(d,r_1,r_2)$
- delete list: $\mathrm{InRoom}(\mathrm{robot},r_1)$, $\mathrm{InRoom}(o,r_1)$
- add list: $\mathrm{InRoom}(\mathrm{robot},r_2)$, $\mathrm{InRoom}(o,r_2)$

STRIPS problem space =
- initial world model, DB_0 (list of ground atoms)
- set of operators (with preconds and effects)
- goal statement (list of atoms)

desired plan: sequence of ground operators

STRIPS Example

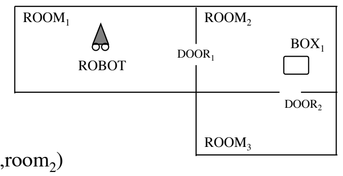
In addition to PushThru, consider

$\mathrm{GoThru}(d,r_1,r_2)$:

precondition: $\mathrm{InRoom}(\mathrm{robot},r_1)$, $\mathrm{Connects}(d,r_1,r_2)$

delete list: $\mathrm{InRoom}(\mathrm{robot},r_1)$

add list: $\mathrm{InRoom}(\mathrm{robot},r_2)$



DB_0:

$\mathrm{InRoom}(\mathrm{robot},\mathrm{room}_1)$ $\mathrm{InRoom}(\mathrm{box}_1,\mathrm{room}_2)$

$\mathrm{Connects}(\mathrm{door}_1,\mathrm{room}_1,\mathrm{room}_2)$ $\mathrm{Box}(\mathrm{box}_1)$

$\mathrm{Connects}(\mathrm{door}_2,\mathrm{room}_2,\mathrm{room}_3)$ …

Goal: $[\ \mathrm{Box}(x) \wedge \mathrm{InRoom}(x,\mathrm{room}_1)\ ]$

Progressive planning

Here is one procedure for planning with a STRIPS like representation:

Input : a world model and a goal
Output : a plan or fail. (ignoring variables)

```
ProgPlan[DB,Goal] =
    If Goal is satisfied in DB, then return empty plan
    For each operator $o$ such that precond($o$) is satisfied in the current DB:
        Let DB´ = DB + addlist($o$) – dellist($o$)
        Let plan = ProgPlan[DB´,Goal]
        If plan $\neq$ fail, then return [act($o$) ; plan]
    End for
    Return fail
```

This depth-first planner searches forward from the given DB_0 for a sequence of operators that eventually satisfies the goal

DB´ is the progressed world state

Regressive planning

Here is another procedure for planning with a STRIPS like representation:

Input : a world model and a goal
Output : a plan or fail.

(ignoring variables)

```
RegrPlan[DB,Goal] =
    If Goal is satisfied in DB, then return empty plan
    For each operator $o$ such that dellist($o$) $\cap$ Goal = {}:
        Let Goal´ = Goal + precond($o$) – addlist($o$)
        Let plan = RegrPlan[DB,Goal´]
        If plan $\neq$ fail, then return [plan ; act($o$)]
    End for
    Return fail
```

This depth-first planner searches backward for a sequence of operators that will reduce the goal to something satisfied in DB_0

Goal´ is the regressed goal

Computational aspects

Even without variables, STRIPS planning is NP-hard.

Many methods have been proposed to avoid redundant search

e.g. partial-order planners, macro operators

One approach: application dependent control

Consider this range of GOLOG programs:

pick an action

< any deterministic program > $\Longrightarrow$ ***while*** $\neg Goal$ ***do*** $\pi a\ .\ a$

fully specific about sequence of actions required	any sequence such that *Goal* holds at end
easy to execute	as hard as planning!

In between, the two extremes we can give domain-dependent guidance to a planner:

while $\neg Goal$ ***do*** πa . [Acceptable$(a)?$; a]

where Acceptable is formalized separately

This is called <u>forward filtering</u> .

Hierarchical planning

The basic mechanisms of planning so far still preserve all detail needed to solve a problem

- attention to too much detail can derail a planner to the point of uselessness
- would be better to first search through an *abstraction space*, where unimportant details were suppressed
- when solution in abstraction space is found, account for remaining details

ABSTRIPS

precondition wffs in abstraction space will have fewer literals than those in ground space

e.g., PushThru operator

- high abstraction: applicable whenever an object is pushable and a door exists
- lower: robot and obj in same room, connected by a door to target room
- lower: door must be open
- original rep: robot next to box, near door

predetermined partial order of predicates with “criticality” level

Reactive systems

Some suggest that explicit, symbolic production of formal plans is something to be avoided (especially considering computational complexity)

even propositional case is intractable; first-order case is undecidable

Just “react”: observe conditions in the world and decide (or look up) *what to do next*

can be more robust in face of unexpected changes in the environment

$\Rightarrow$ <u>reactive systems</u>

“Universal plans”: large lookup table (or boolean circuit) that tells you exactly what to do based on current conditions in the world

Reactive systems have impressive performance on certain low-level problems (e.g. learning to walk), and can even look “intelligent”

but what are the limitations? ...