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:

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

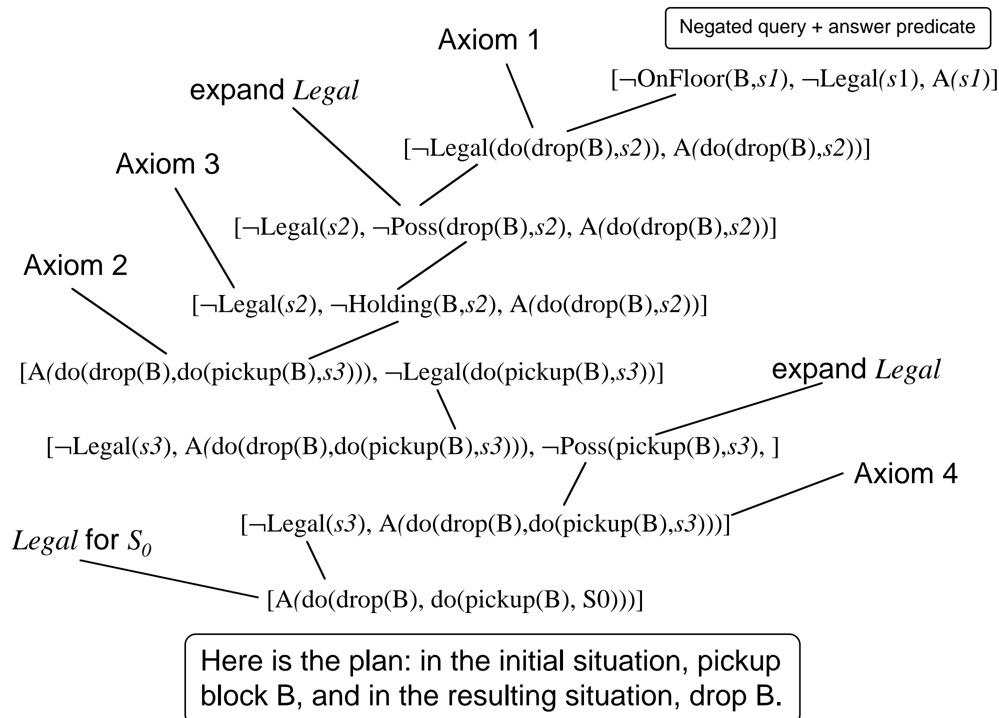
We can see how this will work using a simplified version of a previous example:

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

(Effects, Preconds and Initial state are bracketed as: KB)

Deriving a plan



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

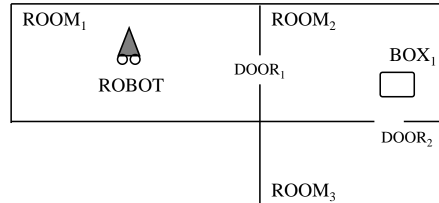
In addition to PushThru, consider

GoThru(d,r_1,r_2):

precondition: InRoom(robot,r_1), Connects(d,r_1,r_2)

delete list: InRoom(robot,r_1)

add list: InRoom(robot,r_2)



DB_0:

InRoom(robot,$room_1$) InRoom(box_1,$room_2$)
Connects($door_1$,$room_1$,$room_2$) Box(box_1)
Connects($door_2$,$room_2$,$room_3$) …

Goal: [Box(x) $\wedge$ InRoom(x,$room_1$)]

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 ≠ 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

 © Brachman & Levesque 2005

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:

< any deterministic program >	→	***while*** $\neg Goal$ ***do*** $\pi a \, . \, a$ (pick an action)
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 forward filtering .

 © Brachman & Levesque 2005

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

⇒ reactive systems

“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? ...