The Logic of Planning

Murray Shanahan

Overview

- Specification and implementation
- Situation calculus
- The frame problem
- Planning in situation calculus
- A Prolog implementation

Notation

- These notes make use of both Prolog and predicate calculus (pure logic). They are closely related but *not* the same thing
  - Prolog is a programming language. Programs in Prolog can be read both *procedurally* (ie: they describe what to do) and *declaratively* (ie: they describe what is true)
  - Predicate calculus is a mathematical formalism that can *only* be read declaratively
- There are several notational differences. Notably, in Prolog, variables begin with upper-case letters, while constants, functions, and predicates begin with lower-case letters. In predicate calculus, it's the *other way around*

$$Happy(x) \leftarrow Student(x) \wedge Clever(x)$$

Predicate calculus

```
happy(X) :- student(X), clever(X)
```

Prolog

Specification and Implementation

- To ensure theoretical rigour, our examination of planning will distinguish specification from implementation
- This approach is good for other cognitive operations as well as planning
- First, we specify in a mathematically precise way what planning is. We'll use predicate calculus to do this
- Second we devise algorithms that conform to the specification
- In this way, we're in a position to prove that the implementation meets the specification, and to compare different implementations for the same specification

What Is Planning?

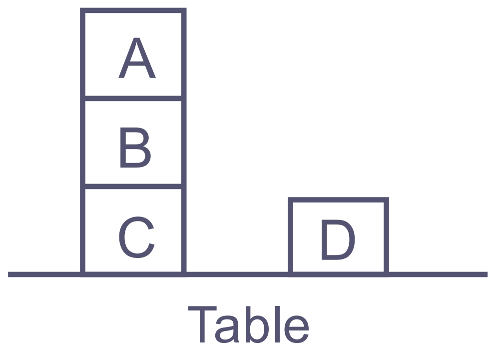
- Given the description Δ of an initial state, the description Γ of a goal state, and a description of the effects of actions, the task of planning is to find a sequence of actions that will transform Δ into Γ
- Finding paths in a graph (as in "Blind Search" and "Informed Search") is an example of this
- But the description of a state in graph search is very simple. It's just the node you're at
- In full-blown planning, states have complex structure. So there's more to designing a planning algorithm than just settling on the search method

The Situation Calculus

- The *situation calculus* is a logic-based formalism for representing the effects of actions
- It is expressed using predicate calculus
- Its ontology (the kinds of things that exist according to the formalism) includes *situations*, *actions*, and *fluents*
- A fluent is something that changes value over time. Actions affect fluents, transforming one situation into another
- A situation can be thought of as a state of affairs, and is characterised by the set of fluents that hold in it
- Using predicate calculus, we'll write *Holds(f,s)* to denote that fluent *f* is true in situation *s*

The Blocks World

- Here's a Blocks World situation, which we'll denote *S0*



- We write *Holds(On(x,y),s)* to denote that block *x* is on block *y* in situation *s*. (Note that *On(x,y)* is a fluent)
- The whole situation is represented with these four formulae

Holds(On(C,Table),S0)
Holds(On(B,C),S0)
Holds(On(A,B),S0)
Holds(On(D,Table),S0)

The *Result* Function

- Let's introduce another fluent. *Clear(x)* holds if *x* has nothing on top of it. So we have:

  Holds(Clear(A),S0)
  Holds(Clear(D),S0)
  Holds(Clear(Table),S0)

- We'll have just one action. Let *Move(x,y)* denote the action of moving *x* onto *y*
- Now for a notational trick. We'll write *Result(a,s)* to denote the situation you get after performing action *a* in situation *s*
- So *Result(Move(A,D),S0)* is the situation you get after moving block *A* onto block *D*, starting in the initial situation
- Nested *Result* terms can capture sequences of actions

  Result(Move(B,A),Result(Move(A,D),S0))

Effect Axioms 1

- We can now write formulae (called *effect axioms*) that describe the effects of actions
- First we'll describe the effects of the *Move* action on the *On* fluent

$$
\begin{aligned}
&Holds(On(x,y),Result(Move(x,y),s)) \leftarrow \\
&\qquad Holds(Clear(x),s) \wedge Holds(Clear(y),s) \wedge x \neq y \wedge x \neq Table
\end{aligned}
$$

$$
\begin{aligned}
&\neg Holds(On(x,z),Result(Move(x,y),s)) \leftarrow \\
&\qquad Holds(Clear(x),s) \wedge Holds(Clear(y),s) \wedge \\
&\qquad Holds(On(x,z),s) \wedge y \neq z \wedge x \neq y
\end{aligned}
$$

- The right hand sides of these formulae take account of the *preconditions* of actions. For example, it is a precondition of the *Move* action that both the block being moved and its destination are clear

Effect Axioms 2

- Next we describe the effects of the *Move* action on the *Clear* fluent

$$
\begin{aligned}
&Holds(Clear(z),Result(Move(x,y),s)) \leftarrow \\
&\qquad Holds(Clear(x),s) \wedge Holds(Clear(y),s) \wedge \\
&\qquad Holds(On(x,z),s) \wedge y \neq z \wedge x \neq y
\end{aligned}
$$

$$
\begin{aligned}
&\neg Holds(Clear(y),Result(Move(x,y),s)) \leftarrow \\
&\qquad Holds(Clear(x),s) \wedge Holds(Clear(y),s) \wedge x \neq y \wedge x \neq Table \wedge \\
&\qquad y \neq Table
\end{aligned}
$$

- Now let
  - Δ be the conjunction of the formulae describing the initial situation, and
  - Σ be the conjunction of the formulae describing the effects of the *Move* action

Frame Axioms

- Now we can prove,

$$\Sigma \wedge \Delta \models Holds(On(A,D),Result(Move(A,D),S0))$$
$$\Sigma \wedge \Delta \models \neg\, Holds(On(A,B),Result(Move(A,D),S0))$$

- This is unsurprising. But we CANNOT prove,

$$\Sigma \wedge \Delta \models Holds(On(B,C),Result(Move(A,D),S0))$$

- This is because we have failed to describe what does *not* change when an action is performed. We can do this explicitly, using *frame axioms*. Here's an example

$$Holds(On(v,w),Result(Move(x,y),s)) \leftarrow Holds(On(v,w),s) \wedge x \neq v$$

More Frame Axioms

- Annoyingly we need loads of frame axioms. We also need,

$$Holds(On(x,y),Result(Move(x,x),s)) \leftarrow Holds(On(x,y),s)$$

which expresses the fact that trying to move something on top of itself has no effect

- And we need,

$$\neg Holds(On(v,w),Result(Move(x,y),s)) \leftarrow$$
$$\neg Holds(On(v,w),s) \wedge [\ x \neq v \vee y \neq w\ ]$$

- We also need a set of frame axioms for the *Clear* fluent.
- In general, for a domain with *n* actions and *m* fluents, we need close to *n* x *m* frame axioms, because most actions leave most fluents unchanged

The Frame Problem

- Let Σ^+ be Σ plus all the necessary frame axioms. Then we will at last be able to show,

$$\Sigma^+ \wedge \Delta \models Holds(On(B,C),Result(Move(A,D),S0))$$

- But we don't want to have to write out all those frame axioms. This is the *frame problem* (as described in 1969 by John McCarthy & Pat Hayes). All we want to say is "*and everything else stays the same*".
- The frame problem (in its full generality) is very tricky. People have written whole books about it:

  Shanahan, M.P. (1997). *Solving the Frame Problem*. MIT Press
- One approach is *non-monotonic reasoning*. More on this later

The Qualification Problem

- Here's another issue. Surely no effect axiom can capture all the preconditions for an action. What about all the weird ways an action might go wrong?
- For example, what if a block is so fragile that it crumbles if a clumsy robot tries to pick it up?

$$Holds(On(x,y),Result(Move(x,y),s)) \leftarrow$$
$$Holds(Clear(x),s) \wedge Holds(Clear(y),s) \wedge x \neq y \wedge x \neq Table \wedge \neg VeryFragile(x)$$

- But how can we anticipate every exception to a rule? This is the *qualification problem*, which is a close relative of the frame problem. We won't propose a solution here, but you should be familiar with the term

The Planning Task

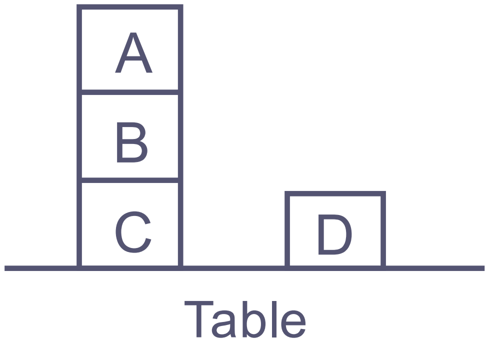
- We're now in a position to give a mathematical specification of situation calculus planning
- Suppose we have an initial situation *S0* and we want to arrive at a goal situation in which fluents f_1 to f_n hold. Let Δ be a formula describing the initial situation, and let Σ^+ be a formula describing the effects of actions (and incorporating a solution to the frame problem)
- Then a plan is a sequence of actions a_1 to a_m such that

$$\Sigma^+ \wedge \Delta \models Holds(f_1,\sigma) \wedge Holds(f_2,\sigma) \wedge \ldots \wedge Holds(f_n,\sigma)$$

where $\sigma = Result(a_m, Result(a_{m-1}, \ldots Result(a_1, S0) \ldots))$

Prolog and Situation Calculus

- Consider the following Blocks World initial situation again



- Using situation calculus, we can express this in Prolog as follows

```
holds(on(c,table),s0).
holds(on(b,c),s0).
holds(on(a,b),s0).
holds(on(d,table),s0).
```

```
holds(clear(a),s0).
holds(clear(d),s0).
holds(clear(table),s0).
```

Effect Axioms in Prolog

- Prolog can represent the Blocks World effect axioms as follows

```
holds(on(X,Y),result(move(X,Y),S)) :-
        holds(clear(X),S), holds(clear(Y),S),
        X≠Y, X≠table.
holds(clear(Z),result(move(X,Y),S)) :-
        holds(clear(X),S), holds(clear(Y),S),
        holds(on(X,Z),S), Y≠Z, X≠Y.
```

- So far, the Prolog implementation matches the pure predicate calculus specification perfectly, which is good
- Note that we don't write negative effect axioms. We cannot write `not(holds(F,S))` on the LHS of a Prolog clause

A Universal Frame Axiom

- Next we have to tackle the frame problem
- To describe the *non-effects* of actions, we could write a set of explicit frame axioms in Prolog. But using negation-as-failure we can do better
- Let `ab(A,F,S)` be true if action `A` changes fluent `F` in situation `S`. (`ab` is short for "abnormal" because most actions leave most fluents unchanged in most situations)
- Now we can write a *universal* frame axiom:

```
holds(F,result(A,S)) :-
        holds(F,S), not ab(A,F,S).
```

Default Reasoning in Prolog

- Note that `not ab(A,F,S)` has a different meaning in Prolog from $\neg$ *Ab(a,f,s)* in predicate calculus. `not p` is true in Prolog if `p` cannot be proved. But to show $\neg$ *P* in predicate calculus, you have to prove it explicitly
- So all we have to do now is describe when `ab` is true, and negation-as-failure will implement the assumption that it is false for all other cases

```
ab(move(X,Y),on(X,Z),S) :-
        holds(clear(X),S), holds(clear(Y),S),
        holds(on(X,Z),S), Y≠Z, X≠Y.
ab(move(X,Y),clear(Y),S) :-
        holds(clear(X),S), holds(clear(Y),S), X≠Y.
```

- This is an example of *default* or *non-monotonic* reasoning

Prediction in Prolog

- Given all the Prolog clauses above, we can use Prolog to do prediction – to reason forwards in time. To see the outcome of moving block A onto block D we can present the following query to Prolog

```
:-  holds(F, result(move(a, d), s0))
```

And we will get the following answers:

```
№1    F = on(a, d)
№2    F = clear(b)
№3    F = on(c, table)
№4    F = on(b, c)
№5    F = on(d, table)
№6    F = clear(a)
№7    F = clear(table)
No more solutions
```

Planning in Prolog

- We can also use logic programming to do planning. But if we present the following query to a normal Prolog interpreter it will loop

```
:-  holds(on(d,b),S),  holds(on(b,a),S)
```

- This is because of Prolog's depth-first search strategy. To make this work we would need a different search strategy, such as breadth-first
- This illustrates the difference between the *general concept* of logic programming, and Prolog, which is just one way to realise that concept
- The following slide shows a possible derivation, with lots of bits omitted

```
holds(on(d,b),S), holds(on(b,a),S)
        |  S=result(move(d,b),S1)
holds(clear(d),S1), holds(clear(b),S1),
holds(on(b,a),S)
        |  S1=result(move(b,a),S2)
holds(clear(d),S1), holds(clear(b),S2),
holds(clear(a),S2)
        |  S2=result(move(a,X),S3)
holds(clear(d),S1), holds(clear(X),S3)
holds(clear(a),S3), holds(clear(a),S2)
        |  S3=s0
        |  X=table
holds(clear(d),S1), holds(clear(a),S3)
holds(clear(a),S2)
        |
        □
```

- Pure Prolog won’ t do this because it always picks the *leftmost* sub-goal to work on next
- But we can write a Prolog *meta-interpreter* do get this behaviour
- A meta-interpreter is a logic programming interpreter written in Prolog
- But we won’t go into further details here

Predicate Completion 1

- The predicate calculus equivalent of this Prolog program is:

$$
\begin{aligned}
&Holds(f,Result(a,s)) \leftrightarrow \\
&\qquad [f=On(x,y) \wedge a=Move(x,y) \wedge \\
&\qquad\qquad Holds(Clear(x),s) \wedge Holds(Clear(y),s) \wedge x\neq y \wedge x\neq Table] \vee \\
&\qquad [f=Clear(z) \wedge a=Move(x,y) \wedge \\
&\qquad\qquad Holds(Clear(x),s) \wedge Holds(Clear(y),s) \wedge \\
&\qquad\qquad\qquad Holds(On(x,z),s) \wedge y\neq z \wedge x\neq y] \vee \\
&\qquad [Holds(f,s) \wedge \neg Ab(a,f,s)]
\end{aligned}
$$

$$
\begin{aligned}
&Ab(a,f,s) \leftrightarrow \\
&\qquad [f=On(x,z) \wedge a=Move(x,y) \wedge \\
&\qquad\qquad Holds(Clear(x),s) \wedge Holds(Clear(y),s) \wedge Holds(On(x,z),s) \wedge \\
&\qquad\qquad\qquad x\neq z \wedge x\neq y] \vee \\
&\qquad [f=Clear(y) \wedge a=Move(x,y) \wedge \\
&\qquad\qquad Holds(Clear(x),s) \wedge Holds(Clear(y),s) \wedge x\neq y]
\end{aligned}
$$

Predicate Completion 2

- The logical formulation on the previous slide is the *predicate completion* Comp(Σ) of the Prolog program Σ
- Predicate completion is one way to supply a semantics for negation-as-failure
- In the case of situation calculus, what we get when we apply predicate completion is a *successor state axiom*
- Successor state axioms are one way to address the frame problem
- DoC's very own Bob Kowalski and Keith Clark solved the frame problem using negation-as-failure in the 1970s. It took the rest of the world a couple of decades to catch up

Non-monotonicity

- Solving the frame problem requires non-monotonicity. Recall that a logic is *monotonic* if, given that

$$\Sigma \models \phi$$

for any ψ, we have

$$\Sigma \wedge \psi \models \phi$$

- In other words, in a monotonic logic, new facts never undermine established conclusions
- Negation-as-failure is non-monotonic. Adding a new clause can make it possible to prove a previously unprovable negated goal
- Consider that Comp($\Sigma \wedge \psi$) is not the same as Comp(Σ) $\wedge\ \psi$