More Planning and Prolog Operators

Artificial Intelligence Programming in Prolog
Lecturer: Tim Smith
Lecture 16
22/11/04
Contents

- Planning continued
  - Implementing MEA
  - Protecting Goals
  - Best-first Planning
  - Partial Order Planning

- Operators
  - What operators do
  - How the built-in operators work
  - How to define your own operators
  - Operators as functors for structures
  - Operators as predicate-names
Welcome to Blocks World

- Blocks World is THE classic Toy-World problem of AI. It has been used to develop AI systems for vision, learning, language understanding, and planning.
- It consists of a set of solid blocks placed on a table top (or, more often, a simulation of a table top). The task is usually to stack the blocks in some predefined order.
- It lends itself well to the planning domain as the rules, and state of the world can be represented simply and clearly.
- Solving simple problems can often prove surprisingly difficult so it provides a robust testing ground for planning systems.
Means-Ends Analysis

- From last lecture….
- **Means-Ends Analysis** plans backwards from the Goal state, generating new states from the preconditions of actions, and checking to see if these are facts in our initial state.

To solve a list of Goals in current state *State*, leading to state *FinalState*, do:

1. If all the Goals are true in *State* then *FinalState* = *State*. Otherwise do the following:
   1. Select a still unsolved Goal from Goals.
   2. Find an Action that adds Goal to the current state.
   3. Enable Action by solving the preconditions of Action, giving *MidState*.
   4. *MidState* is then added as a new Goal to Goals and the program recurses to step 1.
Implementing MEA

[Taken from Bratko, 2001, pg 420]

\[
\text{plan(State,Goals,\text{[]},State)}:-%\text{ Plan is empty}\\
\quad \text{satisfied(State,Goals).-% Goals true in State}
\]

\[
\text{plan(State,Goals,Plan,FinalState)}:-\\
\text{append(PrePlan,[Action|PostPlan],Plan),-% Divide plan}\\
\text{member(Goal,Goals),}\\
\text{\+ member(Goal,State),-% Select a goal}\\
\text{opn(Action,PreCons,Add),}\\
\text{member(Goal,Add),-% Relevant action}\\
\text{can(Action),-% Check action is possible}\\
\text{plan(State,PreCons,PrePlan,MidState1),-% Link Action to Initial State.}\\
\text{apply(MidState1,Action,MidState2),-% Apply Action}\\
\text{plan(MidState2,Goals,PostPlan,FinalState).-% Recurse to link Action to rest of Goals.}
\]
Implementing MEA (2)

\[
\text{opn}(\text{move}(\text{Block, From, To}), \quad \% \text{Name} \\
\quad \text{[clear( Block), clear( To), on( Block, From)]}, \quad \% \text{Precons} \\
\quad \text{[ on(Block,To), clear(From)]}). \quad \% \text{Add List}
\]

\[
\text{can}(\text{move}(<\text{Block, From, To}>)):\-
\quad \text{is\_block(\text{Block}),} \quad \% \text{Block to be moved} \\
\quad \text{object( To),} \quad \% \"To\" \text{is a block or a place} \\
\quad \text{To \It{\neq} Block,} \quad \% \text{Block cannot be moved to itself} \\
\quad \text{object( From),} \quad \% \"From\" \text{is a block or a place} \\
\quad \text{From \It{\neq} To,} \quad \% \text{Move to new position} \\
\quad \text{Block \It{\neq} From.} \quad \% \text{Block not moved from itself}
\]

\[
\text{satisfied}(_,[]). \quad \% \text{All Goals are satisfied} \\
\text{satisfied}(\text{State},[\text{Goal}|\text{Goals}]):- \\
\quad \text{member(Goal,State),} \quad \% \text{Goal is in current State} \\
\quad \text{satisfied}(\text{State},\text{Goals}).
\]

22/11/04
AIPP Lecture 16: More Planning and Operators
Protecting Goals

- MEA produces a very inefficient plan:

\[
\text{Plan} = \text{[move}(b, 3, c), \text{move}(b, c, 3), \text{move}(c, a, 2), \text{move}(a, 1, b), \\
\text{move}(a, b, 1), \text{move}(b, 3, c), \text{move}(a, 1, b)]}
\]
Protecting Goals (2)

- The plan is inefficient as the planner pursues different goals at different times.
- After achieving one goal (e.g. on(b,c) where on(c,a)) it must destroy it in order to achieve another goal (e.g. clear(a)).
- It is difficult to give our planner foresight so that it knows which preconditions are needed to satisfy later goals.
- Instead we can get it to protect goals it has already achieved.
- This forces it to backtrack and find an alternate plan if it finds itself in a situation where it must destroy a previous goal.
- Once a goal is achieved it is added to a Protected list and then every time a new Action is chosen the Action’s delete list is checked to see if it will remove a Protected goal.
Best-first Planning

• So far, our planners have generated plans using *depth-first search* of the space of possible actions.

• As they utilise no domain knowledge when choosing alternative paths, the resulting plans are very inefficient.

• There are three ways *heuristic guidance* could be incorporated:

  1. *The order in which goals are pursued.* For example, when building structures you should always start with the foundations then work up.

  2. *Choosing between actions that achieve the same goal.* You might want to choose an action that achieves as many goals as possible, or has preconditions that are easy to satisfy.

  3. *Evaluating the cost of being in a particular state.*
Best-first Planning (2)

- The state space of a planning problem can be generated by
  - regressing a goal through all the actions that satisfy that goal, and
  - generating all possible sets of sub-goals that satisfy the preconditions for this action.
= this is known as *Goal Regression*.

- The *cost* of each of these sets of sub-goals can then be evaluated.
  - cost = a measure of how difficult it will be to complete a plan when a particular set of goals are left to be satisfied.

- By ordering these sets of sub-goals based on the heuristic evaluation function and *always choosing the ‘cheapest’ set*, our planner can derive a plan using *best-first search*. 
Our current planners will *always consider all possible orderings of actions* even when they are completely independent.

In the above example, the only important factor is that *the two plans do not interact*.

- The order in which moves alternate between plans is unimportant.
- Only the order within plans matters.

Goals can be generated without precedence constraints (e.g. on(a,b), on(d,e)) and then left unordered *unless* later preconditions introduce new constraints (e.g. on(b,c) must precede on(a,b) as \+clear(a)).

= A *Partial-Order Planner* (or Non-Linear Planner).
Summary: Planning

- **Blocks World** is a very common Toy-World problem in AI.
- **Means-Ends Analysis** (MEA) can be used to plan backwards from the Goal state to the Initial state.
  - MEA often creates more direct plans,
  - but is still inefficient as it pursues goals in any order.
- **Goal Protection**: previously completed goals can be protected by making sure that later actions do not destroy them.
  - Forces generation of direct plans through backtracking.
- **Best-first Planning** can use knowledge about the problem domain, the order of actions, and the cost of being in a state to generate the ‘cheapest’ plan.
- **Partial-Order Planning** can be used for problems that contain multiple sets of goals that do not interact.
Part 2: Prolog Operators

*WARNING: The operators discussed in the following slides do not refer to the same “operators” previously seen in planning!* (Just an unfortunate terminological clash)
What is an operator?

- Functors and predicate names generally precede arguments, with arguments grouped using brackets:
  
  \[ \text{ancestor}(\text{fred}, P). \]
  \[ \text{find_age}(\text{person}(\text{fred}, \text{smith}), \text{Age}). \]

- To allow these names to be positioned elsewhere, they have to be declared as **operators**.

- All standard Prolog punctuation and arithmetic symbols are built-in operators.

- An operator can be:
  - **infix** (placed between its arguments) e.g. \( 5 + 6, \ a \ :- \ b, c. \)
  - **prefix** (placed before its arguments, without the need for brackets) e.g. \( \langle + \rangle 5=6, \ -5, \ ?- \ use\_module(X). \)
  - **postfix** (placed after its arguments) e.g. \( 5 \ hr \)

- **THIS IS PURELY A NOTATIONAL CONVENIENCE.**
Operator position

- Usually, an operator can be written in conventional position, and means exactly the same:

  \[ \text{?- X is } +(3, \,(2,4)). \quad \text{?- X is } 3 + 2 * 4. \]
  \[ X = 11 \quad X = 11 \]
  \[ \text{yes} \quad \text{yes} \]
  \[ \text{?- } +(3, \,(2,4)) \text{ = } 3 + 2 * 4. \]
  \[ \text{yes} \]

- For some of the fundamental punctuation symbols of Prolog, such as comma, the name of the operator has to be put in quotes before this can be done:

  \[ \text{?- X = } ',(a, b). \]
  \[ X = a,b \]
  \[ \text{yes} \]

- But it's usually not a good idea to use very basic Prolog notation in non-standard positions.
Operator precedence

- Arithmetic operators obey grouping conventions just as in ordinary algebra/arithmetic.
  \[ a + b * c = a + (b * c) \neq (a + b) * c \]

- In Prolog, operator grouping is controlled by every operator having a precedence, which indicates its priority relative to other operators. (All values shown are for Sicstus Prolog).

- Precedence 500 (both for infix and prefix versions): +, -
- Precedence 400: *, /, // (integer division)
- Precedence 300: mod (the remainder of integer division)

- Operators with lower precedence “stick together” arguments more than those with higher precedence.
- All operators have a precedence between 1200 (for :- and -->) to 200 (for ^).
Associativity

- The associativity of an operator defines how many arguments it takes (its arity) and the grouping of expressions constructed from a series of operators with the same precedence.

- For example, \( a + b + c \) might seem ambiguous between \( a + (b + c) \) and \( (a + b) + c \)

- This makes a difference. For example: \(?- X + Y = a + b + c.\) 
  \[\begin{align*}
  X &= a \\
  Y &= (b+c) \\
  \text{yes}
  \end{align*}\] 
  or this?
  \[\begin{align*}
  X &= (a+b) \\
  Y &= c \\
  \text{yes}
  \end{align*}\]

- This is resolved by defining the functors/operators that are allowed to neighbour a particular operator. The arguments either side and operator (\(f\)) can contain functor/operators that are:
  - of a strictly lower precedence value (notated \(x\)), or
  - of an equal or lower precedence value (notated \(y\)).

- E.g. the “+” sign is “\(y\)” to its left, ”\(x\)” to its right = \(yfx\)
Associativity (2)

- Hence $a + b + c$ must mean $(a + b) + c$, as this makes its left argument be $(a + b)$, whose principal connector is also “$+$”, which is of the same precedence.
- If the right argument were $(b + c)$, that would violate the “strictly lower precedence to the right”

<table>
<thead>
<tr>
<th>$d * a + b \mod c * e + f$ = Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>400 500 300 400 500</td>
</tr>
</tbody>
</table>

$$d * a + (b \mod c) * e + f$$

| 400 500 | 400 500 |

$$(d * a) + (b \mod c) * e + f$$

| 500 | 400 500 |

$$(d * a) + ((b \mod c) * e) + f$$

| 500 | 500 |

$$(d * a) + ((b \mod c) * e) + f$$
Operator definitions

- The Prolog notation for defining an operator uses the predicate `op/3`, with arguments
- the numerical *precedence* (200-1200)
- an expression indicating the *associativity*:
  - infix `{xfx, xfy, yfx, yf}`;
  - prefix `{fx, fy}`;
  - postfix `{xf, yf}`.
- the *name* of the operator, or a list of names for several operators.

So the arithmetical operators are defined as if the system had executed the goals:

- `?- op(500, yfx, [ +, -]).`
- `?- op(500, fx, [ +, - ]).`
- `?- op(400, yfx, [ *, /, //]).`
- `?- op(300, xfx, [ mod ]).`

Any Prolog atom can be declared as a new operator in this way.
Examining operator declarations

- The built-in predicate `current_op/3`, which has the same three arguments as `op/3` can be used to examine any of the current operator declarations.
  
  
  ```prolog
  |?- current_op(Prec, Assoc, +).
  Prec = 500, Assoc = yfx;
  Prec = 500, Assoc = fx;
  no
  ```

  - Note that + is defined as both an infix and prefix operator.

- The built-in predicate `display/1` takes a term as its argument, and displays that term in the conventional form with, all operator positions shown according to their precedence and associativity.
  
  ```prolog
  |?- display(1 + 2 - 3 * 4).
  -(+(1,2),*(3,4))
  yes
  ```
Defining an operator for a structure

- There are two main reasons for defining an operator:
  - as a way of creating a compound structure, or
  - to use a predicate in a non-conventional position.

- Suppose we want a data structure for time-durations, in hours and minutes.
- We could use a structure: \texttt{duration(3, 14)}
  where the 1st component represents hours, 2nd is minutes.
- Or we could make this an infix operator “hr” so our structures would look like: \texttt{3 hr 14}
- The latter is more intuitive to read. Now let us define it.
Precedence

- First we need to choose a precedence level?
- Relative position in hierarchy matters, not exact numerical code.
- If we want to allow
  
  \[ 3 + 2 \text{ hr} \ 5 \times 10 \]

  to be grouped as:

  \[ (3 + 2) \text{ hr} \ (5 \times 10) \]

  then place “hr” higher than the arithmetical operators.
- Defining “hr” lower than the arithmetic operators would interpret it as:

  \[ 3 + (2 \text{ hr} \ 5) \times 10 \]
- Therefore, we will choose somewhere between 500 [+,-] and 550 (the next op upwards) will do.
Choosing associativity

- We only need to consider left- or right-associativity if we want to allow expressions such as:
  
  2 hr 5 hr 10

- As these expressions won’t occur we want our operator to be symmetrically associative.

- Therefore, we should make both sides the same:
  - either both “x” or “y”.

- Definition:
  
  `?- op(525, xfx, hr).` at the command prompt, or
  
  `:- op(525, xfx, hr).` in the consulted file.

- This allows “hr” to be used as an infix operator in Prolog expressions, meaning a structure with functor “hr” and two arguments.
Defining an operator predicate

- The other use of an operator is as a predicate used in a non-conventional position.

- Suppose we wanted to compare our time structures (items with an “hr” operator) for size.

- We can’t just use < or > as the times are recorded as compound structures that need to be deciphered.

- We need to define a suitable-looking operator for this comparison: the “less than” operator could be <<<

- So we want to allow goals such as:

  ?- 3 hr 45 <<< 4 hr 20.

  yes
Defining an operator predicate (2)

• What precedence should it have?
• We want 3 hr 45 <<< 4 hr 20 to be grouped as (3 hr 45) <<< (4 hr 20) so “<<<“ should be higher than “hr”.

• Could put at the same level as the arithmetical comparison operators (<, >, etc.), namely 700.

• Again, no issue regarding associativity. So definition is:

  ?- op(700, xfx, <<<).

  This definition ensures the Prolog system correctly groups expressions containing <<<, but it gives no meaning to the operator!
Giving meaning to an operator

• Once the operator is declared it can be defined as a predicate in the usual way.
  – The head should *exactly* match the format of the intended goal e.g. $H_1 \ hr \ M_1 \ <<\ H_2 \ hr \ M_2$, and
  – The body should carry out the computation and tests necessary to prove the goal as true.

\[
\begin{align*}
H_1 \ hr \ M_1 \ <<\ H_2 \ hr \ M_2 & :\ - \\
\text{\% if hour less, time is less} & \\
H_1 & < H_2. \\

H_1 \ hr \ M_1 \ <<\ H_1 \ hr \ M_2 & :\ - \\
\text{\% hour is same, depends on minutes} & \\
M_1 & < M_2. \\
\end{align*}
\]

?- 3 \ hr \ 50 \ <<\ 4 \ hr \ 10.  
?- 3 \ hr \ 45 \ <<\ 3 \ hr \ 15
yes  
no
Summary: Operators

- Operators can be declared to create
  - novel compound structures, or
  - a predicate in a non-conventional position.

- All operators have:
  - **Precedence**: a value between 200 and 1200 that specifies the grouping of structures made up of more than one operator.
  - **Associativity**: a specification of how structures made up of operators with the same precedence group.

- Operators are defined using `op/3`:
  ```prolog
  :- op(700, xfx, <<<).
  ``

- Once an operator has been defined it can be defined as a predicate in the conventional way.