Planning

Chapter 11

Outline

- Search vs. planning
- STRIPS operators
- Partial-order planning

Search vs. planning

Consider the task get milk, bananas, and a cordless drill
Standard search algorithms seem to fail miserably:

After-the-fact heuristic/goal test inadequate

Search vs. planning contd.

Planning systems do the following:
1) open up action and goal representation to allow selection
2) divide-and-conquer by subgoaling
3) relax requirement for sequential construction of solutions

<table>
<thead>
<tr>
<th>Search</th>
<th>Planning</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>Lisp data structures</td>
</tr>
<tr>
<td>Actions</td>
<td>Lisp code</td>
</tr>
<tr>
<td>Goal</td>
<td>Lisp code</td>
</tr>
<tr>
<td>Plan</td>
<td>Sequence from $s_0$</td>
</tr>
</tbody>
</table>

STRIPS operators

Tidily arranged actions descriptions, restricted language

**Action:** Buy($x$)

**Precondition:** At($p$), Sells($p$, $x$)

**Effect:** Have($x$)

[Note: this abstracts away many important details!]

Restricted language ⇒ efficient algorithm
Precondition: conjunction of positive literals
Effect: conjunction of literals

A complete set of STRIPS operators can be translated into a set of successor-state axioms

Partially ordered plans

*Partially ordered* collection of steps with

- **Start step** has the initial state description as its effect
- **Finish step** has the goal description as its precondition
- causal links from outcome of one step to precondition of another
- temporal ordering between pairs of steps

**Open condition** = precondition of a step not yet causally linked

A plan is **complete** iff every precondition is achieved

A precondition is **achieved** iff it is the effect of an earlier step and no possibly intervening step undoes it
Example

Start
At(Home) Have(Ban.) Have(Drill) Have(Milk)
Sells(SM,Milk) Sells(HWS,Drill) At(Home) Sells(SM,Ban.)

Chapter 11 7

Planning process

Operators on partial plans:
- add a link from an existing action to an open condition
- add a step to fulfill an open condition
- order one step wrt another to remove possible conflicts

Gradually move from incomplete/vague plans to complete, correct plans
Backtrack if an open condition is unachievable or if a conflict is unresolvable

POP algorithm sketch

function POP(initial, goal, operators) returns plan
    plan ← MAKE-MINIMAL-PLAN(initial, goal)
    loop do
        if Solution?(plan) then return plan
        Sneed, c ← SELECT-SUBGOAL(plan)
        CHOOSE-OPERATOR(plan, operators, Sneed, c)
        RESOLVE-THREATS(plan)
    end

function SELECT-SUBGOAL(plan) returns Sneed, c
    pick a plan step Sneed from STEPS(plan)
    with a precondition c that has not been achieved
    return Sneed, c

function CHOOSE-OPERATOR(plan, operators, Sneed, c)
    choose a step Snew from operators or STEPS(plan) that has c as an effect
    if there is no such step then fail
    add the causal link Sneed ←→ Snew to LINKS(plan)
    add the ordering constraint Sneed < Snew to ORDERINGS(plan)
    if Snew is a newly added step from operators then
        add Snew to STEPS(plan)
        add Start ≺ Snew ≺ Finish to ORDERINGS(plan)
end

function RESOLVE-THREATS(plan)
    for each Sthreat that threatens a link Si −→ Sj in LINKS(plan) do
        choose either
        Demotion: Add Sthreat ≺ Si to ORDERINGS(plan)
        Promotion: Add Sj < Sthreat to ORDERINGS(plan)
        if not CONSISTENT(plan) then fail
end

Chapter 11 10

Chapter 11 12
**Clobbering and promotion/demotion**

A clobberer is a potentially intervening step that destroys the condition achieved by a causal link. E.g., Go(Home) clobbers At(Supermarket):

Demotion: put before Go(Supermarket)

Promotion: put after Buy(Milk)

**Properties of POP**

Nondeterministic algorithm: backtracks at choice points on failure:
- choice of $S_{old}$ to achieve $S_{new}$
- choice of demotion or promotion for clobberer
- selection of $S_{new}$ is irrevocable

POP is sound, complete, and systematic (no repetition)

Extensions for disjunction, universals, negation, conditionals
Can be made efficient with good heuristics derived from problem description
Particularly good for problems with many loosely related subgoals

**Example: Blocks world**

"Sussman anomaly" problem

<table>
<thead>
<tr>
<th>Start State</th>
<th>Goal State</th>
</tr>
</thead>
<tbody>
<tr>
<td>B A C</td>
<td>A B C</td>
</tr>
</tbody>
</table>

Clear(x) On(x,z) Clear(y)

PutOn(x,y)

~On(x,z) ~Clear(y) Clear(z) On(x,y)

~On(x,z) Clear(z) On(x,Table)

+ several inequality constraints

**Example contd.**

START

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

Clobber Cl(C)

START

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

PutOn(A,B) Clobber Cl(C) => order after PutOn(B,C)

FINISH
Example contd.

On(A,B) On(B,C) On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

PutOn(B,C) Cl(B) On(B,z) Cl(C)

PutOn(A,B) Cl(A) On(A,z) Cl(B) clobbers Cl(B) => order after PutOn(B,C)

PutOn(B,C) Cl(B) On(B,z) Cl(C) => order after PutOn(A,B) clobbers Cl(B)

PutOnTable(C) Cl(C)On(C,z) PutOn(B,C) clobbers Cl(C) => order after PutOnTable(C)