Planning

- Search vs. planning
- STRIPS operators
- Partial-order planning
What we have so far

• Can TELL KB about new percepts about the world

• KB maintains model of the current world state

• Can ASK KB about any fact that can be inferred from KB

How can we use these components to build a planning agent, i.e., an agent that constructs plans that can achieve its goals, and that then executes these plans?
Example: Robot Manipulators

- Example: (courtesy of Martin Rohrmeier)
  - Puma 560
  - Kr6
Remember: Problem-Solving Agent

Note: This is offline problem-solving. Online problem-solving involves acting w/o complete knowledge of the problem and environment.

```c
function SIMPLE-PROBLEM-SOLVING-AGENT(p) returns an action
inputs: p, a percept
static: s, an action sequence, initially empty
         state, some description of the current world state
         g, a goal, initially null
         problem, a problem formulation

state <- UPDATE-STATE(state, p)
if s is empty then
    g <- FORMULATE-GOAL(state)
    problem <- FORMULATE-PROBLEM(state, g)
    s <- SEARCH(problem)
action <- RECOMMENDATION(s, state)
    s <- REMAINDER(s, state)
return action
```
Simple planning agent

- Use percepts to build model of current world state

- IDEAL-PLANNER: Given a goal, algorithm generates plan of action

- STATE-DESCRIPTION: given percept, return initial state description in format required by planner

- MAKE-GOAL-QUERY: used to ask KB what next goal should be
A Simple Planning Agent

function SIMPLE-PLANNING-AGENT(percept) returns an action

static: KB, a knowledge base (includes action descriptions)
        p, a plan (initially, NoPlan)
        t, a time counter (initially 0)

local variables: G, a goal

        current, a current state description

TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
current ← STATE-DESCRIPTION(KB, t)

if p = NoPlan then
    G ← ASK(KB, MAKE-GOAL-QUERY(t))
    p ← IDEAL-PLANNER(current, G, KB)

if p = NoPlan or p is empty then
    action ← NoOp
else
    action ← FIRST(p)
    p ← REST(p)

    TELL(KB, MAKE-ACTION-SENTENCE(action, t))
t ← t+1

return action

Like popping from a stack
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

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></th>
<th>Search</th>
<th>Planning</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>Lisp data structures</td>
<td>Logical sentences</td>
</tr>
<tr>
<td>Actions</td>
<td>Lisp code</td>
<td>Preconditions/outcomes</td>
</tr>
<tr>
<td>Goal</td>
<td>Lisp code</td>
<td>Logical sentence (conjunction)</td>
</tr>
<tr>
<td>Plan</td>
<td>Sequence from $S_0$</td>
<td>Constraints on actions</td>
</tr>
</tbody>
</table>
Planning in situation calculus

\[ \text{PlanResult}(p, s) \text{ is the situation resulting from executing } p \text{ in } s \]
\[ \text{PlanResult}([], s) = s \]
\[ \text{PlanResult}([a|p], s) = \text{PlanResult}(p, \text{Result}(a, s)) \]

Initial state: \( \text{At(Home, } S_0) \land \neg \text{Have(Milk, } S_0) \land \ldots \)

Actions as Successor State axioms:
\[ \text{Have(Milk, Result}(a, s)) \Leftrightarrow \]
\[ [(a = \text{Buy(Milk)} \land \text{At(Supermarket, } s)) \lor (\text{Have(Milk, } s) \land a \neq \ldots)] \]

Query:
\[ s = \text{PlanResult}(p, S_0) \land \text{At(Home, } s) \land \text{Have(Milk, } s) \land \ldots \]

Solution:
\[ p = [\text{Go(Supermarket)}, \text{Buy(Milk)}, \text{Buy(Bananas)}, \text{Go(HWS)}, \ldots] \]

Principal difficulty: unconstrained branching, hard to apply heuristics
Basic representation for planning

• Most widely used approach: uses STRIPS language

• **states:** conjunctions of function-free ground literals (i.e., predicates applied to constant symbols, possibly negated); e.g.,

\[
\text{At(Home) } \land \lnot \text{Have(Milk)} \land \lnot \text{Have(Bananas)} \land \lnot \text{Have(Drill)} \ldots
\]

• **goals:** also conjunctions of literals; e.g.,

\[
\text{At(Home) } \land \text{Have(Milk)} \land \text{Have(Bananas)} \land \text{Have(Drill)}
\]

but can also contain variables (implicitly universally quant.); e.g.,

\[
\text{At(x) } \land \text{Sells(x, Milk)}
\]
Planner vs. theorem prover

- **Planner**: ask for sequence of actions that makes goal true if executed

- **Theorem prover**: ask whether query sentence is true given KB
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 $\Rightarrow$ efficient algorithm

- Precondition: conjunction of positive literals
- Effect: conjunction of literals

Graphical notation:

\[
\begin{array}{c}
\text{At}(p) \quad \text{Sells}(p,x) \\
\hline
\text{Buy}(x) \\
\hline
\text{Have}(x)
\end{array}
\]
Types of planners

• Situation space planner: search through possible situations

• Progression planner: start with initial state, apply operators until goal is reached
  
  Problem: high branching factor!

• Regression planner: start from goal state and apply operators until start state reached
  
  Why desirable? usually many more operators are applicable to initial state than to goal state.
  
  Difficulty: when want to achieve a conjunction of goals

Initial STRIPS algorithm: situation-space regression planner
State space vs. plan space

Standard search: node = concrete world state
Planning search: node = partial plan

Defn: open condition is a precondition of a step not yet fulfilled

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

Gradually move from incomplete/vague plans to complete, correct plans
Operations on plans

- Refinement operators: add constraints to partial plan

- Modification operator: every other operators
Types of planners

- **Partial order planner**: some steps are ordered, some are not

- **Total order planner**: all steps ordered (thus, plan is a simple list of steps)

- **Linearization**: process of deriving a totally ordered plan from a partially ordered plan.
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
Plan

We formally define a plan as a data structure consisting of:

- Set of plan steps (each is an operator for the problem)
- Set of step ordering constraints
  
  e.g., \( A \prec B \) means “\( A \) before \( B \)”
- Set of variable binding constraints
  
  e.g., \( v = x \) where \( v \) variable and \( x \) constant or other variable
- Set of causal links
  
  e.g., \( A \rightarrow^c B \) means “\( A \) achieves \( c \) for \( B \)”
POP algorithm sketch

function POP(initial, goal, operators) returns plan

    plan ← Make-Minimal-Plan(initial, goal)

loop do

    if Solution?(plan) then return plan

    S_{need}, c ← Select-Subgoal(plan)
    Choose-Operator(plan, operators, S_{need}, c)
    Resolve-Threats(plan)

end

function Select-Subgoal(plan) returns S_{need}, c

    pick a plan step S_{need} from Steps(plan)
    with a precondition c that has not been achieved

    return S_{need}, c
procedure CHOOSE-OPERATOR (plan, operators, S_{need}, c) 

choose a step $S_{add}$ from operators or STEPS(plan) that has $c$ as an effect

if there is no such step then fail

add the causal link $S_{add} \rightarrow S_{need}$ to LINKS(plan)

add the ordering constraint $S_{add} \prec S_{need}$ to ORDERINGS(plan)

if $S_{add}$ is a newly added step from operators then

add $S_{add}$ to STEPS(plan)

add Start $\prec S_{add} \prec$ Finish to ORDERINGS(plan)

---

procedure RESOLVE-THREATS(plan)

for each $S_{threat}$ that threatens a link $S_i \rightarrow S_j$ in LINKS(plan) do

choose either

Demotion: Add $S_{threat} \prec S_i$ to ORDERINGS(plan)

Promotion: Add $S_j \prec S_{threat}$ to ORDERINGS(plan)

if not CONSISTENT(plan) then fail

end

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

Extensions for disjunction, universals, negation, conditionals
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(HWS):

Demotion: put before Go(HWS)

Promotion: put after Buy(Drill)
Example: block world

"Sussman anomaly" problem

Start State

Goal State

\[
\begin{align*}
\text{Clear}(x) \land \text{On}(x,z) \land \text{Clear}(y) & \quad \text{PutOn}(x,y) \\
\neg \text{On}(x,z) \land \neg \text{Clear}(y) & \quad \text{PutOnTable}(x) \\
\text{Clear}(z) \land \text{On}(x,y) & \\
\neg \text{On}(x,z) \land \text{Clear}(z) \land \text{On}(x,\text{Table}) & \\
+ & \text{ several inequality constraints}
\end{align*}
\]
Example (cont.)

START

\[ \text{On}(C, A) \ \text{On}(A, Table) \ \text{Cl}(B) \ \text{On}(B, Table) \ \text{Cl}(C) \]

\[ \text{On}(A, B) \ \text{On}(B, C) \]

FINISH
Example (cont.)
Example (cont.)

\[
\text{START} \\
\text{On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)}
\]

\[
\text{PutOn(A,B)} \\
\text{clobbers Cl(B)} \\
\Rightarrow \text{order after PutOn(B,C)}
\]

\[
\text{Cl(A) On(A,z) Cl(B)} \\
\text{PutOn(A,B)}
\]

\[
\text{Cl(B) On(B,z) Cl(C)} \\
\text{PutOn(B,C)}
\]

\[
\text{On(A,B) On(B,C)} \\
\text{FINISH}
\]
Example (cont.)

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

On(C,z) Cl(C)

PutOnTable(C)

Cl(A) On(A,z) Cl(B)

PutOn(A,B)

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

PutOn(B,C)

On(A,B) On(B,C)

FINISH

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

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