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

```
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
```

Note: This is offline problem-solving. Online problem-solving involves acting w/o complete knowledge of the problem and environment
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)          Like popping from a stack
    p ← REST(p)
TELL(KB, MAKE-ACTION-SENTENCE(action, t))
t ← t + 1
return action
```
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

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

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

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

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

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

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.,

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

• **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.,

\[
\text{At}(x) \land \text{Sells}(x, \text{Milk})
\]
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:** \( \text{Buy}(x) \)
**Precondition:** \( \text{At}(p), \text{Sells}(p, x) \)
**Effect:** \( \text{Have}(x) \)

[Note: this abstracts away many important details!]

Restricted language \( \Rightarrow \) efficient algorithm

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

Graphical notation:

```
At(p)  Sells(p,x)

Buy(x)

Have(x)
```
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., $A \supset B$ 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., $A \xrightarrow{c} B$ means “$A$ achieves $c$ for $B$”
POP algorithm sketch

```plaintext
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
```
POP algorithm (cont.)

**procedure** \texttt{CHOOSE-OPERATOR}(\textit{plan}, \textit{operators}, \textit{S_{need}}, \textit{c})

\begin{itemize}
  \item \texttt{choose} a step \textit{S_{add}} from \textit{operators} or \texttt{STEPS(\textit{plan})} that has \textit{c} as an effect
  \item if there is no such step \texttt{then fail}
  \item add the causal link \textit{S_{add} \rightarrow S_{need}} to \texttt{LINKS(\textit{plan})}
  \item add the ordering constraint \textit{S_{add} \prec S_{need}} to \texttt{ORDERINGS(\textit{plan})}
  \item if \textit{S_{add}} is a newly added step from \textit{operators} \texttt{then}
    \begin{itemize}
      \item add \textit{S_{add}} to \texttt{STEPS(\textit{plan})}
      \item add \textit{Start} \prec \textit{S_{add}} \prec \textit{Finish} to \texttt{ORDERINGS(\textit{plan})}
    \end{itemize}
\end{itemize}

**procedure** \texttt{RESOLVE-THREATS(\textit{plan})}

\begin{itemize}
  \item for each \textit{S_{threat}} that threatens a link \textit{S_i \rightarrow S_j} in \texttt{LINKS(\textit{plan})} \texttt{do}
    \begin{itemize}
      \item \texttt{choose} either
        \begin{itemize}
          \item \textit{Demotion}: Add \textit{S_{threat} \prec S_i} to \texttt{ORDERINGS(\textit{plan})}
          \item \textit{Promotion}: Add \textit{S_j \prec S_{threat}} to \texttt{ORDERINGS(\textit{plan})}
        \end{itemize}
      \item if not \texttt{CONSISTENT(\textit{plan})} \texttt{then fail}
    \end{itemize}
\end{itemize}

POP is sound, complete, and \underline{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

\[
\begin{align*}
\text{Clear}(x) & \quad \text{On}(x,z) \quad \text{Clear}(y) \\
\text{PutOn}(x,y) \\
\sim \text{On}(x,z) & \quad \sim \text{Clear}(y) \\
\text{Clear}(z) & \quad \text{On}(x,y)
\end{align*}
\]

Goal State

\[
\begin{align*}
\text{Clear}(x) & \quad \text{On}(x,z) \\
\text{PutOnTable}(x) \\
\sim \text{On}(x,z) & \quad \text{Clear}(z) \quad \text{On}(x,\text{Table})
\end{align*}
\]

+ several inequality constraints
Example (cont.)

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

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

\[ \text{FINISH} \]
Example (cont.)
Example (cont.)

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