Classical Planning
Partial-Order Planning

Sections 10.1, 10.4.4

Nilufer Onder
Department of Computer Science
Michigan Technological University
Outline

- Search vs. planning
- PDDL operators
- Partial-order planning
Planning is the task of finding a set of actions that will achieve a goal. A planner is a program that searches for a plan. It inputs a description of the world and the goals. The output is a plan. The simplest plan is a sequence of actions: `do action1, do action2 ...` More complex plans may include branching actions: “if (condition) do action1 else do action2”
Tidily arranged actions descriptions, restricted language

\[ \text{At}(s) \neg \text{Bought}(x) \text{ Sells}(s,x) \]

\[ \boxed{\text{BUY } (s,x)} \]

\text{Bought}(x)

\textbf{ACTION: } \text{Buy}(s,x) \\
\textbf{PRECONDITION: } \text{At}(s), \neg \text{Bought}(x), \text{Sells}(s,x) \\
\textbf{EFFECT: } \text{Bought}(x) \]
ACTION: \textit{Buy}(s, x)

PRECONDITION: \textit{At}(s), \neg \textit{Bought}(x), \textit{Sells}(s, x)

EFFECT: \textit{Bought}(x)

- Restricted language $\implies$ efficient algorithm (but many important details will have to be abstracted away)

- Action schema: name, parameters, preconditions, effects

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

- STRIPS is the earliest planning representation
## Search vs. planning (cont’d)

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Planning</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>Data structure</td>
<td>Logical sentences</td>
</tr>
<tr>
<td>Actions</td>
<td>Program</td>
<td>Preconditions/outcomes</td>
</tr>
<tr>
<td>Goal</td>
<td>Program</td>
<td>Logical sentence</td>
</tr>
<tr>
<td></td>
<td>Path from $S_0$</td>
<td>(conjunction)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Sequence of) actions</td>
</tr>
</tbody>
</table>
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
The state of the world is represented by a collection of variables 
(*factored representation*)

Each *state* is represented as a conjunction of fluents that are ground, functionless atoms.

A state is a set (*set semantics*)

Use database semantics, *closed world assumption*: If a fluent is not mentioned, assume it is false.

Fluents that are non-ground, negated, or using functions are not allowed.
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
A partially ordered plan is a 5-tuple \((A, O, C, OC, UL)\)

- \(A\) is the set of actions that make up the plan. They are partially ordered.
- \(O\) is a set of ordering constraints of the form \(A \prec B\). It means \(A\) comes before \(B\).
- \(C\) is the set of causal links in the form \((A, p, B)\) where \(A\) is the *supplier action*, where \(B\) is the *consumer action*, and \(p\) is the condition supplied. It is read as “\(A\) achieves \(p\) for \(B\).”
A partially ordered plan is a 5-tuple \((A, O, C, OC, UL)\)

- \(OC\) is a set of open conditions, i.e., conditions that are not yet supported by causal links. It is of the form \(p \text{ for } A\) where \(p\) is a condition and \(A\) is an action.

- \(UL\) is a set of unsafe links, i.e., causal links whose conditions might be undone by other actions.
A plan is *complete* iff every precondition is achieved, and there are no unsafe links. A precondition is *achieved* iff it is the effect of an earlier step and no *possibly intervening* step undoes it.

In other words, a plan is complete when \( OC \cup UL = \emptyset \).

\( OC \cup UL \) is referred to as the *flaws* in a plan.

When a causal link is established, the corresponding condition is said to be *closed*. 
Example

START

CleanLeftSock

CleanRightSock

OC=
LeftShoeOn for FINISH
RightShoeOn for FINISH

LeftShoeOn

RightShoeOn

FINISH
Example (cont’d)

START

CleanLeftSock  CleanRightSock

LeftSockOn

LEFT SHOE

LeftShoeOn  RightShoeOn

FINISH

OC=
RightShoeOn for FINISH
LeftSockOn for LEFTSHOE
Example (cont’d)

START

CleanLeftSock

LEFT SOCK

LeftSockOn

LEFT SHOE

LeftShoeOn

FINISH

CleanRightSock

RightShoeOn

OC =
CleanLeftSock for LEFTSOCK
RightShoeOn for FINISH
Example (cont’d)

```
START

CleanLeftSock

LEFT SOCK

LeftSockOn

LEFT SHOE

LeftShoeOn

FINISH

CleanRightSock

OC = RightShoeOn for FINISH
```
Example (cont’d)

START

CleanLeftSock

LEFT SOCK

CleanRightSock

RightSockOn

OC = RightSockOn for RIGHTSHOE

LEFT SHOE

LeftSockOn

FINISH

LEFT SHOE

LeftShoeOn

RIGHT SHOE

RightShoeOn
Example (cont’d)

START

CleanLeftSock

LEFT SOCK

LeftSockOn

LEFT SHOE

LeftShoeOn

FINISH

CleanRightSock

RIGHT SOCK

RightSockOn

RIGHT SHOE

RightShoeOn

OC = CleanRightSock for RIGHTSOCK
Example (cont’d)

Finite State Machine Diagram:

- **START**
  - CleanLeftSock
  - CleanRightSock

- **LEFT SOCK**
  - LeftSockOn
  - LeftShoeOn

- **LEFT SHOE**

- **RIGHT SOCK**
  - RightSockOn

- **RIGHT SHOE**

- **FINISH**

**OC**: 
{ }
Planning process

Operators on partial plans:
- close open conditions:
  - add a link from an existing action to an open condition
  - add a step to fulfill an open condition
- resolve threats:
  - 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
**function** Tree-Search (*problem*)
returns a solution, or failure

initialize the frontier using the initial state of *problem*
loop do
  if the frontier is empty then return failure
  choose a leaf node and remove it from the frontier
  if the node contains a goal state
    then return the corresponding solution
  else expand the chosen node and add the resulting nodes to the frontier
end
The initial state, goal state and the operators are given. The planner converts them to required structures.

**Initial state:**

**MAKE-MINIMAL-PLAN** *(initial,goal)*

**Goal-Test:**

**SOLUTION?(plan)**

**SOLUTION?** returns true iff OC and UL are both empty.

**Successor function:**

The successors function could either close an open condition or resolve a threat.
function Successors (plan)
returns a set of partially ordered plans

flaw-type ← Select-Flaw-Type (plan)
if flaw-type is an open condition then
    $S_{need}, c ← Select-Subgoal (plan)$
    return Close-Condition (plan, operators, $S_{need}, c$)
if flaw-type is a threat then
    $S_{threat}, S_i, c, S_j ← Select-Threat(plan)$
    return Resolve-Threat (plan, $S_{threat}, S_i, c, S_j$)
function \textsc{Close-Condition} \( (\text{plan}, \text{operators}, S_{\text{need}}, c) \)
returns \textit{a set of partially ordered plans}

\[
\begin{align*}
\text{plans} & \leftarrow \emptyset \\
\text{for each} \ S_{\text{add}} \text{ from operators or \textsc{Steps}(plan)} \\
\text{that has} \ c \text{ has an effect do} \\
\text{new-plan} & \leftarrow \text{plan} \\
\text{if} \ S_{\text{add}} \text{ is a newly added step from operators then} \\
\text{add} \ S_{\text{add}} \text{ to \textsc{Steps}(new-plan)} \\
\text{add START} & \prec S_{\text{add}} \prec \text{FINISH} \text{ to \textsc{Orderings}(new-plan)} \\
\text{add the causal link} \ (S_{\text{add}}, c, S_{\text{need}}) \text{ to \textsc{Links}(new-plan)} \\
\text{add the ordering constraint} \ (S_{\text{add}} \prec S_{\text{need}}) \text{ to} \\
\text{Orderings (new-plan)} \\
\text{add new-plan to plans} \\
\text{end}
\end{align*}
\]

\text{return} \ new-plans
**function** `RESOLVE-THREAT (plan, S_{threat}, S_i, c, S_j)`  
**returns** a set of partially ordered plans

\[
\begin{align*}
\text{plans} & \leftarrow \emptyset \\
\text{// Demotion:} & \\
\text{new-plan} & \leftarrow \text{plan} \\
\text{add the ordering constraint } (S_{\text{threat}} \prec S_i) \text{ to ORDERINGS (new-plan)} \\
\text{if } \text{new-plan} \text{ is consistent then} & \\
\text{add } \text{new-plan} \text{ to plans} \\
\text{// Promotion:} & \\
\text{new-plan} & \leftarrow \text{plan} \\
\text{add the ordering constraint } (S_j \prec S_{\text{threat}}) \text{ to ORDERINGS (new-plan)} \\
\text{if } \text{new-plan} \text{ is consistent then} & \\
\text{add } \text{new-plan} \text{ to plans} \\
\text{return } \text{new-plans}
\end{align*}
\]
The operators are:
GO (?x, ?y)
preconditions: at(?x)
effects: ~at(?x), at(?y)
BUY (s, i)
preconditions: at(?s), ~bought(?i)
effects: bought(?i)

Agenda:
open subgoals:
bought(A) for f
bought(B) for f
at(H) for f

The subgoals that are currently open are italicized.
add a go(H, J) action (2)

New agenda:
open subgoals:
bought(A) for f
bought(B) for f
at(H) for 2

add a go(J, H) action

supply at(H) for 2 from START
Shopping example (cont’d)

new agenda:
open subgoals:
bought(B) for f
~bought(A) for 3
at(J) for 3

support ~bought(A) from START
new agenda:
open subgoals:
bought(B) for f
at(J) for 3

Support at(J) from GO(H,J)−2
Shopping example (cont’d)

new agenda:  open subgoals:    ~bought(B) for 4

new agenda:    bought(A) for 3

new agenda:  open subgoals:    bought(B) for f

HAVEN’T CONSIDERED THE THREATS YET!
Now, the solution is a possible ordering of this plan. Those are:

2 3 4 1
2 3 1 4
2 4 3 1
2 4 1 3
2 1 3 4
2 1 4 3

It should not be possible to order GO(J,H) before any of the BUY actions.
This is a correct partially ordered plan. It is complete.
The possible total orders are:
2 3 4 1
2 4 3 1

The agent has to go to Jim’s first.
It order of getting the items does not matter.
Then it has to go back home.
A **threatening step** is a potentially intervening step that destroys the condition achieved by a causal link. E.g., GO(J,H) threatens At(J)

Demotion: put before GO(H,J)

Promotion: put BUY(Apples)
Properties of POP

- Nondeterministic algorithm: backtracks at choice points on failure:
  - choice of $S_{add}$ to achieve $S_{need}$
  - choice of demotion or promotion for threat resolution
  - selection of $S_{need}$ is irrevocable

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

- Extensions for disjunction, universals, negation, conditionals

- Particularly good for problems with many loosely related subgoals
The flat tire example shows the effect of inserting an “impossible” action.

The Sussman anomaly shows that “divide-and-conquer” is not always optimal. POP can find the optimal plan.
The flat tire domain

Init(At(Flat,Axle) \land At(Spare,Trunk))
Goal(At(Spare,Axle))
Action(REMOVE(spare, trunk),
  Precond: At(spare, trunk)
  Effect: \neg At(spare, trunk) \land At(spare, ground)
Action(REMOVE(flat, axle),
  Precond: At(flat, axle)
  Effect: \neg At(flat, axle) \land At(flat, ground)
Action(PUTON(spare, axle),
  Precond: At(spare, ground) \land \neg at(flat, axle)
  Effect: \neg At(spare, ground) \land At(spare, axle)
Action(LEAVEOVERNIGHT
  Precond: 
  Effect: \neg At(spare, ground) \land \neg At(spare, axle)
  \neg At(spare, trunk) \land \neg At(flat, ground)
  \neg At(flat, axle)
The flat tire plan

START
at(spare, trunk)
at(flat, axle)

at(spare, ground)

REMOVE(spare, trunk)

PUTON(spare, axle)

FINISH
at(spare, axle)
The flat tire plan (cont’d)

START
at(spare,trunk) at(flat,axle)

LEAVEOVERNIGHT

at(spare,ground)
~at(flat,axle)
~at(flat,ground)
~at(spare,axle)
~at(spare,ground)
~at(spare,trunk)

REMOVE(spare,trunk)

PUTON(spare,axle) at(spare,axle) FINISH

at(spare,ground)
~at(flat,axle)
~at(flat,ground)
~at(spare,axle)
~at(spare,ground)
~at(spare,trunk)
The flat tire plan (cont’d)

START

at(spare, trunk)

REMOVE(spare, trunk)

at(spare, trunk)

at(spare, ground)

~at(flat, axle)

PUTON(spare, axle)

at(spare, axle) FINISH

at(flat, axle)

REMOVE(flat, axle)

at(flat, axle)
Clear(x)  On(x,z)  Clear(y)

\[\text{PUTON}(x,y)\]

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

Clear(z)  On(x,y)

Clear(x)  On(x,z)

\[\text{PUTONTABLE}(x)\]

~On(x,z)

Clear(z)  On(x,\text{Table})

+ several inequality constraints
Sussman anomaly (cont’d)

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

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

START

FINISH
If we try the first goal (on(A,B)) first, we can’t proceed without undoing work.
If we try the second goal (on (B,C)) first, we can’t proceed without undoing work.

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

START

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

FINISH
Sussman anomaly (cont’d)

START
On(C,A)  On(A,Table)  Clear(B)  On(B,Table)  Clear(C)

Clear(B)  On(B,z)  Clear(C)
PUTON(B,C)

On(A,B)  On(B,C)
FINISH
Sussman anomaly (cont’d)

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

Clear(A)  On(A,z)  Clear(B)

PUTON(A,B)

Clear(B)  On(B,z)  Clear(C)

PUTON(B,C)

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

START

FINISH
Sussman anomaly (cont’d)

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

On(C,z)  Clear(C)

PUTONTABLE(C)

Clear(A)  On(A,z)  Clear(B)

PUTON(A,B)

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

START

FINISH
Heuristics for POP

POP be made efficient with good heuristics derived from problem description

- Which plan to select?
- Which flaw to choose?
- (We will see more after planning graphs)
Sources for the slides

- AIMA textbook (3rd edition)
- AIMA slides (http://aima.cs.berkeley.edu/)