Planning

- Representations for classical planning
- Modern heuristics for state-space planning
- Planning graphs: a modern planning technique

Background reading

Automated Planning by Malik Ghallab, Dana Nau, Paolo Traverso, Morgan Kaufmann 2004. Chapters 1, 2, 4 & 6

Slides designed by Michael Thielscher
Some Dictionary Definitions of “Plan”

**plan n.**

1. A scheme, program, or method worked out beforehand for the accomplishment of an objective: a *plan of attack.*

2. A proposed or tentative project or course of action: *had no plans for the evening.*

[a representation] of future behaviour … usually a set of actions, with temporal and other constraints on them, for execution by some agent or agents.

– Austin Tate, *MIT Encyclopaedia of the Cognitive Sciences, 1999*
Planning for an Agent/Robot in a Dynamic World

- S is an abstraction that deals only with the aspects that the planner needs to reason about

State transition system
\[ S = (S, A, g) \]
- \( S = \{\text{states}\} \)
- \( A = \{\text{actions}\} \)
- \( g = \text{state-transition function} \)
Example

Example $\mathcal{A} = (S,A,\mathcal{R})$:

- $S = \{s_0, \ldots, s_5\}$
- $A = \{\text{move1, move2, put, take, load, unload}\}$
- $\mathcal{R}$: see the arrows

Dock Worker Robots (DWR) example
Example

- **Classical plan**: a sequence of actions
  \(\langle \text{take, move1, load, move2} \rangle\)

Dock Worker Robots (DWR) example
Domain-Specific Planners

- Many successful real-world planning systems work this way
  - Mars exploration, sheet-metal bending, playing bridge, etc.
- Often use problem-specific techniques that are difficult to generalise to other planning domains
Domain-Independent Planners

- No domain-specific knowledge except the description of the system
- In practice,
  - Not feasible to make domain-independent planners work well in all possible planning domains
- Make simplifying assumptions to restrict the set of domains
  - Classical planning
    - Historical focus of most research on automated planning
Classical Planning

- Reduces to the following problem:
  - Given \( s \), initial state \( s_0 \), and goal states \( S_g \),
  - find a sequence of actions \( (a_1, a_2, \ldots, a_n) \) that produces
  - a sequence of state transitions \( (s_0, s_1, s_2, \ldots, s_n) \) such that \( s_n \in S_g \)

Is this trivial?

- Generalise the earlier example:
  - Five locations, three robot carts, 100 containers, three piles
  - \( 10^{277} \) states

- Automated-planning research has been heavily dominated by classical planning. There are dozens of different algorithms.
Representations for Classical Planning
Classical Representations: Motivation

- In most problems, far too many states to try to represent all of them explicitly as $s_0, s_1, s_2, \ldots$
  - represent each state as a set of \textit{atomic features}

- Define a set of \textbf{operators} that can be used to compute state-transitions
- Don’t give all of the states explicitly
  - Just give the initial state
  - Use the operators to generate the other states as needed
Classical Representation

- Language of first-order logic but without function symbols
  - finitely many predicate symbols and constant symbols

- Example: the DWR domain
  - Locations: l₁, l₂, ...
  - Containers: c₁, c₂, ...
  - Piles: p₁, p₂, ...
  - Robot carts: r₁, r₂, ...
  - Cranes: k₁, k₂, ...
Example (cont'd)

- **Fixed relations**: same in all states
  - adjacent($l,l'$)  attached($p,l$)  belong($k,l$)

- **Dynamic relations**: differ from one state to another
  - occupied($l$)  at($r,l$)
  - loaded($r,c$)  unloaded($r$)
  - holding($k,c$)  empty($k$)
  - in($c,p$)  on($c,c'$)
  - top($c,p$)  top(pallet,$p$)

- **Actions**:
  - take($c,k,p$)  put($c,k,p$)
  - load($r,c,k$)  unload($r$)  move($r,l,l'$)
A **state** is a set of ground atoms

- The atoms represent the things that can be true in some states
- Only finitely many ground atoms, so only finitely many possible states

\[
s_1 = \{\text{attached}(p1,\text{loc1}), \text{in}(c1,p1), \text{in}(c3,p1), \text{top}(c3,p1), \\
\text{on}(c3,c1), \text{on}(c1,\text{pallet}), \text{attached}(p2,\text{loc1}), \text{in}(c2,p2), \\
\text{top}(c2,p2), \text{on}(c2,\text{pallet}), \text{belong}(\text{crane1},\text{loc1}), \\
\text{empty}(\text{crane1}), \text{adjacent}(\text{loc1},\text{loc2}), \text{adjacent}(\text{loc2},\text{loc1}), \\
\text{at}(r1,\text{loc2}), \text{occupied}(\text{loc2}), \text{unloaded}(r1)\}\n\]
Operators

An operator is a triple $o = (\text{name}(o), \text{precond}(o), \text{effects}(o))$

- **name($o$):** a syntactic expression of the form $n(x_1,\ldots,x_k)$
  - $(x_1,\ldots,x_k)$ is a list of every variable symbol (parameter) that appears in $o$
- **precond($o$):** **preconditions**
  - literals that must be true in order to use the operator
- **effects($o$):** **effects**
  - literals the operator will make true

**Example**

```
take(k,l,c,d,p)
  ;; crane k at location l takes c off of d in pile p
precond: belong(k,l), attached(p,l), empty(k), top(c,p), on(c,d)
effects: holding(k,c), ¬empty(k), ¬in(c,p), ¬top(c,p), ¬on(c,d), top(d,p)
```
Actions

An **action** is a ground instance (via a substitution) of an operator

\[
\text{take}(k,l,c,d,p) \\
\text{;; crane } k \text{ at location } l \text{ takes } c \text{ off of } d \text{ in pile } p \\
\text{precond: } \text{belong}(k,l), \text{attached}(p,l), \text{empty}(k), \text{top}(c,p), \text{on}(c,d) \\
\text{effects: } \text{holding}(k,c), \neg \text{empty}(k), \neg \text{in}(c,p), \neg \text{top}(c,p), \neg \text{on}(c,d), \text{top}(d,p)
\]

- Let \( \sigma = \{k/crane1, l/loc1, c/c3, d/c1, p/p1\} \)
- Then \( \text{take}(k,l,c,d,p)\sigma \) is the following action:

\[
\text{take}(\text{crane1}, \text{loc1}, c3, c1, p1) \\
\text{precond: } \text{belong}(\text{crane1}, \text{loc1}), \text{attached}(p1, \text{loc1}), \text{empty}(\text{crane1}), \text{top}(c3,p1), \text{on}(c3,c1) \\
\text{effects: } \text{holding}(\text{crane1},c3), \neg \text{empty}(\text{crane1}), \neg \text{in}(c3,p1), \neg \text{top}(c3,p1), \neg \text{on}(c3,c1), \text{top}(c1,p1)
\]
Applicability and Result of Actions

Let $S$ be a set of literals. Then

$S^+ = \{\text{atoms that appear positively in } S\}$
$S^- = \{\text{atoms that appear negatively in } S\}$

Let $a$ be an operator or action. Then

$\text{precond}^+(a) = \{\text{atoms that appear positively in } a\text{'s preconditions}\}$
$\text{precond}^-(a) = \{\text{atoms that appear negatively in } a\text{'s preconditions}\}$
$\text{effects}^+(a) = \{\text{atoms that appear positively in } a\text{'s effects}\}$
$\text{effects}^-(a) = \{\text{atoms that appear negatively in } a\text{'s effects}\}$

Action $a$ is **applicable** to (or **executable** in) $S$ if

- $\text{precond}^+(a) \subseteq S$
- $\text{precond}^-(a) \cap S = \emptyset$

The **result** of applying action $a$ to state $S$ is

$y(s,a) = (s \setminus \text{effects}^-(a)) \cup \text{effects}^+(a)$
Example: Applicability

An action:

\[ \text{take}(\text{crane1}, \text{loc1}, \text{c3}, \text{c1}, \text{p1}) \]

precond: \begin{align*}
& \text{belong}(\text{crane}, \text{loc1}), \\
& \text{attached}(\text{p1}, \text{loc1}), \\
& \text{empty}(\text{crane1}), \\
& \text{top}(\text{c3}, \text{p1}), \\
& \text{on}(\text{c3}, \text{c1})
\end{align*}

effects: \begin{align*}
& \text{holding}(\text{crane1}, \text{c3}), \\
& \neg \text{empty}(\text{crane1}), \\
& \neg \text{in}(\text{c3}, \text{p1}), \\
& \neg \text{top}(\text{c3}, \text{p1}), \\
& \neg \text{on}(\text{c3}, \text{c1}), \\
& \text{top}(\text{c1}, \text{p1})
\end{align*}

A state it’s applicable to

\[ s_1 = \{ \text{attached}(\text{p1}, \text{loc1}), \text{in}(\text{c1}, \text{p1}), \text{in}(\text{c3}, \text{p1}), \text{top}(\text{c3}, \text{p1}), \text{on}(\text{c3}, \text{c1}), \text{on}(\text{c1}, \text{pallet}), \text{attached}(\text{p2}, \text{loc1}), \text{in}(\text{c2}, \text{p2}), \text{top}(\text{c2}, \text{p2}), \text{on}(\text{c2}, \text{pallet}), \text{belong}(\text{crane1}, \text{loc1}), \text{empty}(\text{crane1}), \text{adjacent}(\text{loc1}, \text{loc2}), \text{adjacent}(\text{loc2}, \text{loc1}), \text{at}(\text{r1}, \text{loc2}), \text{occupied}(\text{loc2}, \text{unloaded}(\text{r1})) \} \]
Example: Result

take(crane1, loc1, c3, c1, p1)

precond:  
belong(crane, loc1),  
attached(p1, loc1),  
empty(crane1), top(c3, p1),  
on(c3, c1)

effects:  
holding(crane1, c3),  
¬empty(crane1),  
¬in(c3, p1), ¬top(c3, p1),  
¬on(c3, c1), top(c1, p1)

s_2 = {attached(p1, loc1), in(c1, p1), in(c3, p1),  
top(c3, p1), on(c3, c1), on(c1, pallet),  
attached(p2, loc1), in(c2, p2),  
top(c2, p2), on(c2, pallet),  
belong(crane1, loc1), empty(crane1),  
adjacent(loc1, loc2),  
adjacent(loc2, loc1), at(r1, loc2),  
occupied(loc2, unloaded(r1),  
holding(crane1, c3), top(c1, p1)}
Exercise
Exercise: The Blocks World

- Infinitely wide table, finite number of children’s blocks
- Ignore where a block is located on the table
- A block can sit on the table or on another block
- There’s a robot gripper that can hold at most one block

- Want to move blocks from one configuration to another
  - e.g.,

  ![Diagram of initial and goal states](image)

  initial state  goal
  c  d  a  b  e  a  b  c
Exercise: Classical Representation – Symbols

- Constant symbols:
  - The blocks: a, b, c, d, e
- Dynamic relations?
Exercise: Classical Operators

- Preconditions and effects?
Summary: Planning Problems

Given a planning domain (language $L$, operators $O$)

- **Representation** of a planning problem: a triple $P = (O, s_0, g)$
  - $O$ is the collection of operators
  - $s_0$ is a state (the initial state)
  - $g$ is a set of literals (the goal formula)
Let $P = (O, s_0, g)$ be a planning problem

- **Plan**: any sequence of actions $\pi = \langle a_1, a_2, \ldots, a_n \rangle$ such that each $a_i$ is an instance of an operator in $O$
- Plan $\pi$ is a **solution** for $P = (O, s_0, g)$ if it is executable and achieves $g$
  
  i.e., if there are states $s_0, s_1, \ldots, s_n$ such that
  
  \[
  \gamma(s_0, a_1) = s_1 \\
  \gamma(s_1, a_2) = s_2 \\
  \vdots \\
  \gamma(s_{n-1}, a_n) = s_n \\
  s_n \text{ satisfies } g
  \]
Example: The 5 DWR Operators

\text{move}(r, l, m)
\begin{itemize}
\item \text{precond:} adjacent(l, m), \text{at}(r, l), \neg \text{occupied}(m)
\item \text{effects:} \text{at}(r, m), \text{occupied}(m), \neg \text{occupied}(l), \neg \text{at}(r, l)
\end{itemize}

\text{load}(k, l, c, r)
\begin{itemize}
\item \text{precond:} \text{belong}(k, l), \text{holding}(k, c), \text{at}(r, l), \text{unloaded}(r)
\item \text{effects:} \text{empty}(k), \neg \text{holding}(k, c), \text{loaded}(r, c), \neg \text{unloaded}(r)
\end{itemize}

\text{unload}(k, l, c, r)
\begin{itemize}
\item \text{precond:} \text{belong}(k, l), \text{at}(r, l), \text{loaded}(r, c), \text{empty}(k)
\item \text{effects:} \neg \text{empty}(k), \text{holding}(k, c), \text{unloaded}(r), \neg \text{loaded}(r, c)
\end{itemize}

\text{put}(k, l, c, d, p)
\begin{itemize}
\item \text{precond:} \text{belong}(k, l), \text{attached}(p, l), \text{holding}(k, c), \text{top}(d, p)
\item \text{effects:} \neg \text{holding}(k, c), \text{empty}(k), \text{in}(c, p), \text{top}(c, p), \text{on}(c, d), \neg \text{top}(d, p)
\end{itemize}

\text{take}(k, l, c, d, p)
\begin{itemize}
\item \text{precond:} \text{belong}(k, l), \text{attached}(p, l), \text{empty}(k), \text{top}(c, p), \text{on}(c, d)
\item \text{effects:} \text{holding}(k, c), \neg \text{empty}(k), \neg \text{in}(c, p), \neg \text{top}(c, p), \neg \text{on}(c, d), \text{top}(d, p)
\end{itemize}
Example

- Let $P = (O, s_0, g)$, where
  - $O = \{\text{the 5 DWR operators}\}$
  - $s_0 = \{\text{attached}(p1,\text{loc1}), \text{in}(c1,p1), \text{in}(c3,p1), \text{top}(c3,p1), \text{on}(c3,c1), \text{on}(c1,\text{pallet}), \text{attached}(p2,\text{loc1}), \text{in}(c2,p2), \text{top}(c2,p2), \text{on}(c2,\text{pallet}), \text{belong}(\text{crane1},\text{loc1}), \text{empty}(\text{crane1}), \text{adjacent}(\text{loc1},\text{loc2}), \text{adjacent}(\text{loc2},\text{loc1}), \text{at}(r1,\text{loc2}), \text{occupied}(\text{loc2}), \text{unloaded}(r1)\}$
  - $g = \{\text{loaded}(r1,c3), \text{at}(r1,\text{loc2})\}$
Two redundant solutions (can remove actions and still have a solution):

\[
\langle \text{move(r1,loc2,loc1)}, \text{take(crane1,loc1,c3,c1,p1)}, \text{move(r1,loc1,loc2)}, \text{move(r1,loc2,loc1)}, \text{load(crane1,loc1,c3,r1)}, \text{move(r1,loc1,loc2)} \rangle
\]

\[
\langle \text{take(crane1,loc1,c3,c1,p1)}, \text{put(crane1,loc1,c3,c2,p2)}, \text{move(r1,loc2,loc1)}, \text{take(crane1,loc1,c3,c2,p2)}, \text{load(crane1,loc1,c3,r1)}, \text{move(r1,loc1,loc2)} \rangle
\]

A solution that is both irredundant and shortest:

\[
\langle \text{move(r1,loc2,loc1)}, \text{take(crane1,loc1,c3,c1,p1)}, \text{load(crane1,loc1,c3,r1)}, \text{move(r1,loc1,loc2)} \rangle
\]

Are there any other shortest solutions? Are irredundant solutions always shortest?
Exercise
Exercise: Plans

Solution?
State-Variable Representation

- Use ground atoms for properties that do not change, e.g., adjacent(loc1,loc2)
- For properties that can change, assign values to state variables
  - Like fields in a record structure

\[
\text{move}(r, l, m) \\
\quad \text{;; robot } r \text{ at location } l \text{ moves to an adjacent location } m \\
\text{precond: } \text{rloc}(r) = l, \text{adjacent}(l, m) \\
\text{effects: } \text{rloc}(r) \leftarrow m
\]

\[
s_1 = \{ \text{top}(p1)=c3, \text{cpos}(c3)=c1, \text{cpos}(c1)=\text{pallet}, \text{holding}(\text{crane1})=\text{nil}, \text{rloc}(r1)=\text{loc2}, \text{loaded}(r1)=\text{nil}, ... \}
\]
Expressive Power

- Any problem that can be represented in one representation can also be represented in the other
- Can convert in linear time and space

$P(x_1, \ldots, x_n)$ becomes
\[ f_P(x_1, \ldots, x_n) = 1 \]

$f(x_1, \ldots, x_n) = y$ becomes
\[ P_f(x_1, \ldots, x_n, y) \]
Comparison

- Classical representation
  - The most popular for classical planning, partly for historical reasons

- State-variable representation
  - Equivalent to classical representation in expressive power
  - Less natural for logicians, more natural for engineers and most computer scientists
  - Useful in non-classical planning problems as a way to handle numbers, functions, time
State-Space Planning
Search Algorithms

Search tree
- nodes = states
- edges = actions
Search Algorithms

Search tree
- nodes = states
- edges = actions

Most common search method: **depth-first** search
- In general, sound but not complete
  - But classical planning has only finitely many states
    - can make depth-first search complete by doing loop-checking
Exercise
Exercise: Interchange Values of Variables

- Operator $assign(v,w,x,y)$
  - precond: $value(v,x), value(w,y)$
  - effects: $\neg value(v,x), value(v,y)$

- Initial state $s_0 = \{ value(a,3), value(b,5), value(c,0) \}$
- Goal $g = \{ value(a,5), value(b,3) \}$

In the search tree for this planning problem,

- what is the length of the shortest path to a solution?
- what is the length of the longest path in the tree?
Planning with Heuristic Search

- Explicitly search with heuristic $h(s)$ that estimates cost from $s$ to goal

- General idea:
  
  heuristic function = length of optimal plan for a \textit{relaxed problem}

- Example:
  
  - Manhattan distance in 15-puzzle

- How to get such heuristics automatically?
General-Purpose Heuristics for Classical Planning

- Automatic extraction of informative heuristic function **from the problem P itself**

- Most common relaxation in planning: **ignore all negative effects** of the operators.

Let $P^+$ be obtained from planning problem $P$ by dropping the negative effects. If $c^*(P^+,s)$ is optimal cost of $P^+$ with initial state $s$, then the heuristic is set to

$$h(s) = c^*(P^+,s)$$

- This heuristic is intractable in general, but easy to approximate

**Example.**
- Operator $\text{assign}(v,w,x,y)$
  - precond: $\text{value}(v,x), \text{value}(w,y)$
  - effects: $\neg\text{value}(v,x), \text{value}(v,y)$

- $s_0 = \{ \text{value}(a,3), \text{value}(b,5), \text{value}(c,0) \}$, $g = \{ \text{value}(a,5), \text{value}(b,3) \}$

- Optimal relaxed plan: $\text{assign}(a,b,3,5), \text{assign}(b,a,5,3)$, hence $h(s_0) = 2$
Example

Operator \texttt{assign}(v,w,x,y)

- \texttt{precond}: value(v,x), value(w,y)
- \texttt{effects}: \neg value(v,x), value(v,y)

\[ g = \{ \text{value}(a,5), \text{value}(b,3) \} \]

\[ s_0 = \{ \text{value}(a,3), \text{value}(b,5), \text{value}(c,0) \} \]

Consider all possible successor states after one action:

\[ s_1 = \{ \text{value}(a,5), \text{value}(b,5), \text{value}(c,0) \} \quad h(s_1) = \infty \]
\[ s_2 = \{ \text{value}(a,3), \text{value}(b,3), \text{value}(c,0) \} \quad h(s_2) = \infty \]
\[ s_3 = \{ \text{value}(a,0), \text{value}(b,5), \text{value}(c,0) \} \quad h(s_3) = \infty \]
\[ s_4 = \{ \text{value}(a,3), \text{value}(b,5), \text{value}(c,3) \} \quad h(s_4) = 2 \]
\[ s_5 = \{ \text{value}(a,3), \text{value}(b,0), \text{value}(c,0) \} \quad h(s_5) = \infty \]
\[ s_6 = \{ \text{value}(a,3), \text{value}(b,5), \text{value}(c,5) \} \quad h(s_6) = 2 \]

No relaxed plan exists
Example

- Operator \( \text{assign}(v, w, x, y) \)
  - precond: \( \text{value}(v, x), \text{value}(w, y) \)
  - effects: \( \neg \text{value}(v, x), \text{value}(v, y) \)

- \( g = \{ \text{value}(a, 5), \text{value}(b, 3) \} \)

- \( s_4 = \{ \text{value}(a, 3), \text{value}(b, 5), \text{value}(c, 3) \} \)

Consider all possible successor states after next action:

\( s_7 = \{ \text{value}(a, 5), \text{value}(b, 5), \text{value}(c, 3) \} \quad h(s_1) = 1 \)

\( s_8 = \{ \text{value}(a, 3), \text{value}(b, 3), \text{value}(c, 3) \} \quad h(s_8) = \infty \)

\( s_9 = \{ \text{value}(a, 3), \text{value}(b, 5), \text{value}(c, 5) \} \quad h(s_9) = 2 \)

One of the successor states of \( s_7 \) is a goal state:

\( s_{10} = \{ \text{value}(a, 5), \text{value}(b, 3), \text{value}(c, 3) \} \)
Planning-Graph Techniques
History

- Before Graphplan came out, most planning researchers were working on Plan Space Search-like planners

- **Graphplan** caused a sensation because it was so much faster

- Many subsequent planning systems have used ideas from it
  - IPP, STAN, GraphHTN, SGP, Blackbox, Medic, TGP, LPG
  - Many of them even much faster than the original Graphplan
Motivation

- A standard tree search may try lots of actions that are unrelated to the goal.

- One way to reduce branching factor:
  - First create a relaxed problem
    - Remove some restrictions of the original problem
      - Want the relaxed problem to be easy to solve (polynomial time)
    - The solutions to the relaxed problem will include all solutions to the original problem
  - Then do a modified version of the original search
    - Restrict its search space to include only those actions that occur in solutions to the relaxed problem.

[Diagram showing tree search with states and actions, including nodes labeled s0, s1, s2, s3, s4, s5, and sg with actions a1, a2, a3, a4, a5, and arrows connecting them.]
Graphplan

procedure Graphplan:

- for $k = 0, 1, 2, \ldots$

  - **Graph expansion:**
    - create a “planning graph” that contains $k$ “levels”
    - Check whether the planning graph satisfies a necessary (but insufficient) condition for plan existence

- If it does, then

  - do **solution extraction:**
    - backward search, modified to consider only the actions in the planning graph
    - if we find a solution, then return it
Example: Have the Cake and Eat it Too

<table>
<thead>
<tr>
<th>Operator Name</th>
<th>Preconditions</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>eat(c)</td>
<td>have(c)</td>
<td>¬have(c), eaten(c)</td>
</tr>
<tr>
<td>bake(c)</td>
<td>¬have(c)</td>
<td>have(c)</td>
</tr>
</tbody>
</table>

Also have the maintenance actions: one for each literal

- $s_0 = \{ \text{have(cake)} \}$
- $g = \{ \text{have(cake), eaten(cake)} \}$

state-level 0

have(cake)

¬eaten(cake)
Example: Have the Cake and Eat it Too

<table>
<thead>
<tr>
<th>Operator Name</th>
<th>Preconditions</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>eat(c)</td>
<td>have(c)</td>
<td>¬have(c), eaten(c)</td>
</tr>
<tr>
<td>bake(c)</td>
<td>¬have(c)</td>
<td>have(c)</td>
</tr>
</tbody>
</table>

Also have the maintenance actions: one for each literal

- s0 = { have(cake) }
- g = { have(cake), eaten(cake) }
Example: Have the Cake and Eat it Too

<table>
<thead>
<tr>
<th>Operator Name</th>
<th>Preconditions</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>eat(c)</td>
<td>have(c)</td>
<td>¬have(c), eaten(c)</td>
</tr>
<tr>
<td>bake(c)</td>
<td>¬have(c)</td>
<td>have(c)</td>
</tr>
</tbody>
</table>

Also have the maintenance actions: one for each literal

s0 = { have(cake) }  
g = { have(cake), eaten(cake) }

"mutex": actions cannot occur together
Example: Have the Cake and Eat it Too

<table>
<thead>
<tr>
<th>Operator Name</th>
<th>Preconditions</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>eat(c)</td>
<td>have(c)</td>
<td>¬have(c), eaten(c)</td>
</tr>
<tr>
<td>bake(c)</td>
<td>¬have(c)</td>
<td>have(c)</td>
</tr>
</tbody>
</table>

Also have the maintenance actions: one for each literal

- s0 = { have(cake) }
- g = { have(cake), eaten(cake) }

"mutex": fluents cannot be obtained together
Example: Have the Cake and Eat it Too

<table>
<thead>
<tr>
<th>Operator Name</th>
<th>Preconditions</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>eat(c)</td>
<td>have(c)</td>
<td>¬have(c), eaten(c)</td>
</tr>
<tr>
<td>bake(c)</td>
<td>¬have(c)</td>
<td>have(c)</td>
</tr>
</tbody>
</table>

Also have the maintenance actions: one for each literal

- s0 = { have(cake) }
- g = { have(cake), eaten(cake) }

Solution extraction **not** called since goals are mutex
Example: Have the Cake and Eat it Too

<table>
<thead>
<tr>
<th>Operator Name</th>
<th>Preconditions</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>eat(c)</td>
<td>have(c)</td>
<td>¬have(c), eaten(c)</td>
</tr>
<tr>
<td>bake(c)</td>
<td>¬have(c)</td>
<td>have(c)</td>
</tr>
</tbody>
</table>

Also have the maintenance actions: one for each literal

- s0 = { have(cake) }
- g = { have(cake), eaten(cake) }
Example: Have the Cake and Eat it Too

- **Operator Name**
- **Preconditions**
- **Effects**

<table>
<thead>
<tr>
<th>Operator Name</th>
<th>Preconditions</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>eat(c)</td>
<td>have(c)</td>
<td>¬have(c), eaten(c)</td>
</tr>
<tr>
<td>bake(c)</td>
<td>¬have(c)</td>
<td>have(c)</td>
</tr>
</tbody>
</table>

Also have the maintenance actions: one for each literal

s0 = { have(cake) }  
g = { have(cake), eaten(cake) }

State-level 0  
Action-level 1  
State-level 1  
Action-level 2  
State-level 2
Example: Have the Cake and Eat it Too

<table>
<thead>
<tr>
<th>Operator Name</th>
<th>Preconditions</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>eat(c)</td>
<td>have(c)</td>
<td>¬have(c), eaten(c)</td>
</tr>
<tr>
<td>bake(c)</td>
<td>¬have(c)</td>
<td>have(c)</td>
</tr>
</tbody>
</table>

Also have the maintenance actions: one for each literal

s0 = { have(cake) }
g = { have(cake), eaten(cake) }
The Planning Graph

- Search space for a relaxed version of the planning problem
- Alternating layers of ground literals and actions
  - Nodes at action-level $i$: actions that might be possible to execute at time $i$
  - Nodes at state-level $i$: literals that might possibly be true at time $i$
- Edges: preconditions and effects

**Maintenance** action: for the case where a literal remains unchanged
Mutual Exclusion

- Two actions at the same action-level are mutex if
  1. Inconsistent effects: an effect of one negates an effect of the other
  2. Interference: one deletes a precondition of the other
  3. Competing needs: they have mutually exclusive preconditions

- Otherwise they don’t interfere with each other
  - Both may appear in a solution plan
  - Two literals at the same state-level are mutex if
    4. Inconsistent support: one is the negation of the other, or all ways of achieving them are pairwise mutex
## Mutexes in the Cake-Example

<table>
<thead>
<tr>
<th>Level</th>
<th>Mutexes</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>\textit{eat(cake)} \hspace{1em} \text{mhave(cake)}</td>
<td>1 (also 2)</td>
</tr>
<tr>
<td>A1</td>
<td>\textit{eat(cake)} \hspace{1em} \text{n\hspace{1em}eaten(cake)}</td>
<td>1, 2</td>
</tr>
<tr>
<td>S1</td>
<td>\textit{have(cake)} \hspace{1em} \text{n\hspace{1em}have(cake)}</td>
<td>4</td>
</tr>
<tr>
<td>S1</td>
<td>\textit{eaten(cake)} \hspace{1em} \text{n\hspace{1em}eaten(cake)}</td>
<td>4</td>
</tr>
<tr>
<td>S1</td>
<td>\textit{have(cake)} \hspace{1em} \text{eaten(cake)}</td>
<td>4</td>
</tr>
<tr>
<td>S1</td>
<td>\text{n\hspace{1em}have(cake)} \hspace{1em} \text{n\hspace{1em}eaten(cake)}</td>
<td>4</td>
</tr>
<tr>
<td>A2</td>
<td>\textit{bake(cake)} \hspace{1em} \text{eat(cake)}</td>
<td>1, 3</td>
</tr>
<tr>
<td>A2</td>
<td>\textit{bake(cake)} \hspace{1em} \text{m\hspace{1em}have(cake)}</td>
<td>1, 2</td>
</tr>
<tr>
<td>A2</td>
<td>\textit{bake(cake)} \hspace{1em} \text{m\hspace{1em}n\hspace{1em}have(cake)}</td>
<td>1, 2</td>
</tr>
<tr>
<td>A2</td>
<td>\textit{bake(cake)} \hspace{1em} \text{m\hspace{1em}have(cake)}</td>
<td>2</td>
</tr>
<tr>
<td>A2</td>
<td>\textit{eat(cake)} \hspace{1em} \text{m\hspace{1em}have(cake)}</td>
<td>1, 2</td>
</tr>
<tr>
<td>A2</td>
<td>\textit{eat(cake)} \hspace{1em} \text{m\hspace{1em}n\hspace{1em}have(cake)}</td>
<td>2, 3</td>
</tr>
<tr>
<td>A2</td>
<td>\textit{eat(cake)} \hspace{1em} \text{m\hspace{1em}n\hspace{1em}eaten(cake)}</td>
<td>3</td>
</tr>
<tr>
<td>A2</td>
<td>\textit{m\hspace{1em}have(cake)} \hspace{1em} \text{m\hspace{1em}have(cake)}</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>A2</td>
<td>\textit{m\hspace{1em}eaten(cake)} \hspace{1em} \text{m\hspace{1em}n\hspace{1em}eaten(cake)}</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>S2</td>
<td>\textit{have(cake)} \hspace{1em} \text{n\hspace{1em}have(cake)}</td>
<td>4</td>
</tr>
<tr>
<td>S2</td>
<td>\textit{eaten(cake)} \hspace{1em} \text{n\hspace{1em}eaten(cake)}</td>
<td>4</td>
</tr>
<tr>
<td>S2</td>
<td>\text{n\hspace{1em}have(cake)} \hspace{1em} \text{n\hspace{1em}eaten(cake)}</td>
<td>4</td>
</tr>
</tbody>
</table>
Example: Have the Cake and Eat it Too

Solution extraction succeeds
( = plan without mutexes)
procedure Solution-extraction(g, j)
    if j = 0 then return the solution
    for each literal l in g
        nondeterministically choose an action to use in state $s_{j-1}$ to achieve l
        if any pair of chosen actions are mutex then backtrack
        $g' := \{\text{the preconditions of the chosen actions}\}$
        Solution-extraction($g'$, j–1)
    end Solution-extraction
Comparison with State-Space Planning

**Advantage:**
- The backward-search part (solution extraction) of Graphplan—which is the hard part—will only look at the actions in the planning graph smaller search space than state-space planning; thus faster

**Disadvantage:**
- To generate the planning graph, Graphplan creates a huge number of ground atoms
  - Many of them may be irrelevant

For classical planning, the advantage outweighs the disadvantage
- GraphPlan solves classical planning problems much faster than SSP without heuristics
Summary

- Representations for classical planning
  - Classical representation
  - State-variable representation

- State-space planning
  - with heuristics

- Planning graphs
  - Creating the graph
  - Adding mutexes
  - Searching the graph