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.

- These two are closest to the meaning used in AI

3. A systematic arrangement of elements or important parts; a configuration or outline: a seating plan; the plan of a story.
4. A drawing or diagram made to scale showing the structure or arrangement of something.
5. A program or policy stipulating a service or benefit: a pension plan.
[a representation] of future behavior … usually a set of actions, with temporal and other constraints on them, for execution by some agent or agents. - Austin Tate

[MIT Encyclopedia of the Cognitive Sciences, 1999]

<table>
<thead>
<tr>
<th>Process Code</th>
<th>Machine</th>
<th>Time (min)</th>
<th>Cycle Time (min)</th>
<th>Action Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>VMC1</td>
<td>2.00</td>
<td>0.00</td>
<td>Orient board</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Clamp board</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Establish datum point at bullseye (0.25, 1.00)</td>
</tr>
<tr>
<td>001</td>
<td>VMC1</td>
<td>0.10</td>
<td>0.43</td>
<td>Install 0.30-diameter drill bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rough drill at (1.25, -0.50) to depth 1.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Finish drill at (1.25, -0.50) to depth 1.00</td>
</tr>
<tr>
<td>001</td>
<td>VMC1</td>
<td>0.10</td>
<td>0.77</td>
<td>Install 0.20-diameter drill bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rough drill at (0.00, 4.88) to depth 1.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Finish drill at (0.00, 4.88) to depth 1.00</td>
</tr>
<tr>
<td>004</td>
<td>VMC1</td>
<td>2.20</td>
<td>1.20</td>
<td>Total time on VMC1</td>
</tr>
<tr>
<td>004</td>
<td>VMC1</td>
<td>0.10</td>
<td>0.34</td>
<td>Install 0.15-diameter side-milling tool</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rough side-mill pocket at (-0.25, 1.25) length 0.40, width 0.30, depth 0.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Finish side-mill pocket at (-0.25, 1.25) length 0.40, width 0.30, depth 0.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rough side-mill pocket at (-0.25, 3.00) length 0.40, width 0.30, depth 0.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Finish side-mill pocket at (-0.25, 3.00) length 0.40, width 0.30, depth 0.50</td>
</tr>
<tr>
<td>004</td>
<td>VMC1</td>
<td>0.10</td>
<td>1.54</td>
<td>Install 0.08-diameter end-milling tool</td>
</tr>
<tr>
<td>004</td>
<td>VMC1</td>
<td>2.50</td>
<td>4.87</td>
<td>Total time on VMC1</td>
</tr>
<tr>
<td>005</td>
<td>EC1</td>
<td>0.00</td>
<td>32.29</td>
<td>Pre-clean board (scrub and wash)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Dry board in oven at 85 deg. F</td>
</tr>
<tr>
<td>005</td>
<td>EC1</td>
<td>30.00</td>
<td>0.48</td>
<td>Setup</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Spread photoresist from 18000 RPM spinner</td>
</tr>
<tr>
<td>005</td>
<td>EC1</td>
<td>30.00</td>
<td>2.00</td>
<td>Setup</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Photolithography of photoresist using phototool in “real.iges”</td>
</tr>
<tr>
<td>005</td>
<td>EC1</td>
<td>30.00</td>
<td>20.00</td>
<td>Setup</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Etching of copper</td>
</tr>
<tr>
<td>005</td>
<td>EC1</td>
<td>30.00</td>
<td>54.77</td>
<td>Setup</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Total time on EC1</td>
</tr>
<tr>
<td>006</td>
<td>MC1</td>
<td>30.00</td>
<td>4.57</td>
<td>Setup</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Prepare board for soldering</td>
</tr>
<tr>
<td>006</td>
<td>MC1</td>
<td>30.00</td>
<td>7.50</td>
<td>Setup</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Screenprint solder stop on board</td>
</tr>
<tr>
<td>006</td>
<td>MC1</td>
<td>30.00</td>
<td>18.00</td>
<td>Setup</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Total time on MC1</td>
</tr>
<tr>
<td>011</td>
<td>TC1</td>
<td>0.00</td>
<td>35.00</td>
<td>Perform post-cap testing on board</td>
</tr>
<tr>
<td>011</td>
<td>TC1</td>
<td>0.00</td>
<td>29.67</td>
<td>Perform final inspection of board</td>
</tr>
</tbody>
</table>

Total time to manufacture: 319.70 minutes
Manufacturing

- Sheet-metal bending machines - Amada Corporation
  - Software to plan the sequence of bends
    [Gupta and Bourne, *J. Manufacturing Sci. and Engr.*, 1999]
Space Exploration

- Autonomous planning, scheduling, control
  - NASA: JPL and Ames
- Remote Agent Experiment (RAX)
  - Deep Space 1
- Mars Exploration Rover (MER)
On January 26th, 2274 Mars days into the mission, NASA declared Spirit a 'stationary research station', expected to stay operational for several more months until the dust buildup on its solar panels forces a final shutdown.
On January 26th, 2274 Mars days into the mission, NASA declared Spirit a 'stationary research station', expected to stay operational for several more months until the dust buildup on its solar panels forces a final shutdown.
Outline

- Conceptual model for planning
- Restrictive assumptions to simplify the problem
- Classical planning
Source Material

- My lectures on AI planning are based partly on Russell & Norvig, and partly on following book:

  - M. Ghallab, D. Nau, and P. Traverso
    *Automated Planning: Theory and Practice*
    Morgan Kaufmann Publishers
    May 2004
    - Web site: http://www.laas.fr/planning

- For CMSC 421, you *don’t* need this book
  - The lecture slides are self-contained
Conceptual Model

1. Environment

State transition system

\[ \Sigma = (S, A, E, \gamma) \]

- \( S = \{ \text{states} \} \)
- \( A = \{ \text{actions} \} \)
- \( E = \{ \text{exogenous events} \} \)
- \( \gamma = \text{state-transition function} \)
Example: The Blocks World

- Infinitely wide table, finite number of children’s blocks
- A robot hand that can pick up blocks and put them down
- A block can sit on the table or on another block
- Ignore where the blocks are located on the table
- Just consider:
  - whether each block is on the table, on another block, or being held
  - whether each block is clear or covered by another block
  - whether the robot hand is holding anything

Example state of the world:

For $n$ blocks, the number of states is more than $n!$
State Transition System

$$\Sigma = (S,A,E,\gamma)$$

- **S** = \{states\}
- **A** = \{actions\}
- **E** = \{exogenous events\}
- **State-transition function**

\[ \gamma: S \times (A \cup E) \rightarrow 2^S \]

- **S** = \{s_0, s_1, s_2, \ldots, s_{22}\}
- **A** = \{take c off of a, put c on the table, …\}
- **E** = \{\}
- **\gamma**: see the arrows
Observation function $h: S \rightarrow O$

**Conceptual Model**

2. **Controller**

- **Initial state**
- **Objectives**

**Execution status**

- **Planner**
  - **Description of $\Sigma$**
  - **Plans**
- **Controller**
  - **Observations**
  - **Actions**
- **System $\Sigma$**
  - **Events**

Given observation $o$ in $O$, produces action $a$ in $A$
Conceptual Model

3. Planner’s Input

Planning problem

Omit unless planning is online

Initial state
Objectives

Execution status

Planner

Description of Σ

Controller

Observations
Actions

System Σ

Plans

Events
Planning Problem

A planning problem includes:

- A description of $\Sigma$
- An initial state, e.g., $s_0$
  - or a set of possible initial states (maybe with a probability distribution)
- An objective, e.g.,
  - a goal state, e.g., $s_4$
  - a set of goal states, e.g.,
    - {all states in which $b$ is on $a$}
  - a task to perform, e.g.,
    - put all the blocks into a single stack
  - a “trajectory” of states
  - an objective function
  - …
Conceptual Model
4. Planner’s Output

Instruction to the controller
Plans

- **Classical plan:**
  a sequence of actions
  
  (take c off of a, put c on the table, take b off the table, put b on a)

- **Policy:**
  a partial function from $S$ into $A$
  
  $\{(s_0, \text{take c off of a}), (s_1, \text{put c on the table}), (s_2, \text{take b off the table}), (s_3, \text{put b on a})\}$
Planning Versus Scheduling

- **Scheduling**
  - Decide when and how to perform a given set of actions
    - Time constraints
    - Resource constraints
    - Objective functions
  - Typically NP-complete

- **Planning**
  - Decide what actions to use to achieve some set of objectives
  - Can be much worse than NP-complete
    - worst case is undecidable
Three Main Types of Planners

1. Domain-specific
2. Domain-independent
3. Configurable

I’ll talk briefly about each
1. Domain-Specific Planners (Chapters 19-23)

- Made or tuned for a specific domain
- Won’t work well (if at all) in any other domain
- Most successful real-world planning systems work this way
Types of Planners
2. Domain-Independent

- In principle, a domain-independent planner works in any planning domain
- Uses no domain-specific knowledge except the definitions of the basic actions
2. Domain-Independent

- In practice,
  - Not feasible to develop domain-independent planners that work in every possible domain

- Make simplifying assumptions to restrict the set of domains
  - *Classical planning*
  - Historical focus of most automated-planning research
Restrictive Assumptions

- **A0: Finite system:**
  - finitely many states, actions, events

- **A1: Fully observable:**
  - the controller always $\Sigma$’s current state

- **A2: Deterministic:**
  - each action has only one outcome

- **A3: Static** (no exogenous events):
  - no changes but the controller’s actions

- **A4: Attainment goals:**
  - a set of goal states $S_g$

- **A5: Sequential plans:**
  - a plan is a linearly ordered sequence of actions ($a_1, a_2, \ldots a_n$)

- **A6: Implicit time:**
  - no time durations; linear sequence of instantaneous states

- **A7: Off-line planning:**
  - planner doesn’t know the execution status
Classical Planning (Chapters 2-9)

- Classical planning requires all eight restrictive assumptions
  - Offline generation of action sequences for a deterministic, static, finite system, with complete knowledge, attainment goals, and implicit time
- Reduces to a search problem:
  - Given $(\Sigma, s_0, S_g)$
    - $s_0$ is the initial state, $S_g$ is a set of goal states
  - Find a sequence of actions $(a_1, a_2, \ldots a_n)$ that produces a sequence of state transitions $(s_1, s_2, \ldots, s_n)$ such that $s_n$ is in $S_g$.
- Constraint-satisfaction problems also were search problems
  - But there were special-purpose problem representations and algorithms that were much faster than ordinary search algorithms
- Can do something similar for planning problems
  - Several ways to do this
  - I’ll discuss a few of the better-known ones
Problem Representation

- Several ways to represent classical planning domains
  - The *classical representation* (or *STRIPS representation*) is the best known
- That’s what I’ll describe
Symbols

- Start with a *function-free* first-order language
  - Finitely many predicate names and constant symbols, infinitely many variable symbols, but *no* function symbols
  - Add a finite set of *operator names*
- e.g., symbols for the blocks world:
  - Constant symbols: a, b, c, d, e, … (names of blocks)
  - Variable symbols: u, v, w, x, y, z, x₁, x₂, …
  - Predicates:
    - `ontable(x)` - block x is on the table
    - `on(x,y)` - block x is on block y
    - `clear(x)` - block x has nothing on it
    - `holding(x)` - the robot hand is holding block x
    - `handempty` - the robot hand isn’t holding anything
  - Operator names: pickup, putdown, stack, unstack
States

- State: a set $s$ of ground atoms representing what’s currently true
- Only finitely many ground atoms, so only finitely many possible states

Example:

\{ontable(a), on(c,a), clear(c), ontable(b), clear(b), holding(d), ontable(e), clear(e)\}
Operators

- **Operator**: a triple (head, preconditions, effects)
  - head: an operator name and a parameter list
    - E.g., opname($x_1$, …, $x_k$)
    - No two operators can have the same name
    - Parameter list must include *all* of the operator’s variables
  - preconditions: literals that must be true to use the operator
  - effects: literals that the operator will make true

- We’ll generally write operators in the following form:

  - **opname($x_1$, …, $x_k$)**
    - Precond: $p_1$, $p_2$, …, $p_m$
    - Effects: $e_1$, $e_2$, …, $e_n$
Blocks-World Operators

unstack\((x, y)\)
- Precond: on\((x, y)\), clear\((x)\), handempty
- Effects: ~on\((x, y)\), ~clear\((x)\), ~handempty, holding\((x)\), clear\((y)\)

stack\((x, y)\)
- Precond: holding\((x)\), clear\((y)\)
- Effects: ~holding\((x)\), ~clear\((y)\), on\((x, y)\), clear\((x)\), handempty

pickup\((x)\)
- Precond: ontable\((x)\), clear\((x)\), handempty
- Effects: ~ontable\((x)\), ~clear\((x)\), ~handempty, holding\((x)\)

putdown\((x)\)
- Precond: holding\((x)\)
- Effects: ~holding\((x)\), ontable\((x)\), clear\((x)\), handempty
Actions and Plans

- Action: a ground instance (via substitution) of an operator

unstack\((x, y)\)

Precond: on\((x, y)\), clear\((x)\), handempty

Effects: \(\neg \text{on}(x, y)\), \(\neg \text{clear}(x)\), \(\neg \text{handempty}\), holding\((x)\), clear\((y)\)

unstack\((c, a)\)

Precond: on\((c, a)\), clear\((c)\), handempty

Effects: \(\neg \text{on}(c, a)\), \(\neg \text{clear}(c)\), \(\neg \text{handempty}\), holding\((c)\), clear\((a)\)
Notation

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

- Example:
  - $\text{unstack}(x,y)$
    - Precond: $\text{on}(x,y), \text{clear}(x), \text{handempty}$
    - Effects: $\sim\text{on}(x,y), \sim\text{clear}(x), \sim\text{handempty}, \text{holding}(x), \text{clear}(y)$
  - $\text{effects}^+(\text{unstack}(x,y)) = \{\text{holding}(x), \text{clear}(y)\}$
  - $\text{effects}^-(\text{unstack}(x,y)) = \{\text{on}(x,y), \text{clear}(x), \text{handempty}\}$
Executability

- An action $a$ is *executable* in $s$ if $s$ satisfies precond($a$),
  - i.e., if $\text{precond}^+(a) \subseteq s$ and $\text{precond}^-(a) \cap s = \emptyset$
- An operator $o$ is *applicable* to $s$ if there’s a ground instance $a$ of $o$ that is executable in $s$
- Example:
  - $s = \{\text{ontable}(a), \text{on}(c,a), \text{clear}(c), \text{ontable}(b), \text{clear}(b), \text{handempty}\}$
  - $o = \text{unstack}(x,y)$
  - $a = \text{unstack}(c,a)$

  **unstack($x,y$)**
  
  Precond: $\text{on}(x,y), \text{clear}(x), \text{handempty}$
  
  Effects: $\neg\text{on}(x,y), \neg\text{clear}(x), \neg\text{handempty}, \text{holding}(x), \text{clear}(y)$

  **unstack($c,a$)**
  
  Precond: $\text{on}(c,a), \text{clear}(c), \text{handempty}$
  
  Effects: $\neg\text{on}(c,a), \neg\text{clear}(c), \neg\text{handempty}, \text{holding}(c), \text{clear}(a)$
Result of performing an action

- If $a$ is executable in $s$, the result of performing it is
  \[ \gamma(s,a) = (s - \text{effects}^{-}(a)) \cup \text{effects}^{+}(a) \]
  - Delete the negative effects, and add the positive ones

- $s = \{\text{ontable}(a), \text{on}(c,a), \text{clear}(c), \text{ontable}(b), \text{clear}(b), \text{handempty}\}$

- $a = \text{unstack}(c,a)$
  - Precond: $\text{on}(c,a), \text{clear}(c), \text{handempty}$
  - Effects: $\sim\text{on}(c,a), \sim\text{clear}(c), \sim\text{handempty}$, holding($c$), clear($a$)

- $\gamma(s,a) = \{\text{ontable}(a), \text{on}(c,a), \text{clear}(c), \text{ontable}(b), \text{clear}(b), \text{handempty}$, holding($c$), clear($a$)\}
Executability of Plans

- Plan: a sequence of actions \( \pi = (a_1, \ldots, a_n) \)
- A plan \( \pi = (a_1, \ldots, a_n) \) is *executable* in the state \( s_0 \) if
  - \( a_1 \) is executable in \( s_0 \), producing some state \( s_1 = \gamma(s_0, a_1) \)
  - \( a_2 \) is executable in \( s_1 \), producing some state \( s_2 = \gamma(s_1, a_2) \)
  - …
  - \( a_n \) is executable in \( s_{n-1} \), producing some state \( s_n = \gamma(s_{n-1}, a_n) \)
- In this case, we define \( \gamma(s_0, \pi) = s_n \)
- Example on next slide
\( s = \{ \text{ontable}(a), \text{on}(c,a), \text{clear}(c), \text{ontable}(b), \text{clear}(b), \text{handempty} \} \)
\( \pi = (\text{unstack}(c,a), \text{putdown}(c), \text{pickup}(b), \text{stack}(b,a)) \)

**unstack\((c,a)\)**
- Precond: \(\text{on}(c,a), \text{clear}(c), \text{handempty} \)
- Effects: \(\neg\text{on}(c,a), \neg\text{clear}(c), \neg\text{handempty}, \text{holding}(c), \text{clear}(a) \)

**putdown\((c)\)**
- Precond: \(\text{holding}(c) \)
- Effects: \(\neg\text{holding}(c), \text{ontable}(c), \text{clear}(c), \text{handempty} \)

**pickup\((b)\)**
- Precond: \(\text{ontable}(b), \text{clear}(b), \text{handempty} \)
- Effects: \(\neg\text{ontable}(b), \neg\text{clear}(b), \neg\text{handempty}, \text{holding}(b) \)

**stack\((b,a)\)**
- Precond: \(\text{holding}(b), \text{clear}(a) \)
- Effects: \(\neg\text{holding}(b), \neg\text{clear}(a), \text{on}(b,a), \text{clear}(b), \text{handempty} \)
Problems and Solutions

- **Planning problem**: a triple $P = (O, s_0, g)$
  - $O$ is a set of operators
  - $s_0$ is the *initial state* - a set of atoms
  - $g$ the *goal formula* - a set of literals

- Every state that satisfies $g$ is a *goal state*

- A plan $\pi$ is a *solution* for $P = (O, s_0, g)$ if
  - $\pi$ is executable in $s_0$
  - the resulting state $\gamma(s_0, \pi)$ satisfies $g$
Example

- $O = \{\text{stack}(x,y), \text{unstack}(x,y), \text{pickup}(x), \text{putdown}(x)\}$

- $s_0 = \{\text{ontable}(a), \text{on}(c,a), \text{clear}(c), \text{ontable}(b), \text{clear}(b), \text{handempty}\}$

- $g = \{\text{on}(a,b)\}$

- One of the solutions is
  - $\pi = (\text{unstack}(c,a), \text{putdown}(c), \text{pickup}(a), \text{stack}(a,b))$
Forward-Search Algorithms

- Go forward from the initial state

- Breadth-first and best-first
  - **Sound**: if they return a plan, then the plan is a solution
  - **Complete**: if a problem has a solution, then they will return one
  - Usually not practical because they require too much memory
    » Memory requirement is exponential in the length of the solution

- Depth-first search, greedy search
  - More practical to use
  - Worst-case memory requirement is linear in the length of the solution
  - Sound but not complete

- But classical planning has only finitely many states
  - Thus, can make depth-first search complete by doing loop-checking
Branching Factor of Forward Search

- Forward search can have a very large branching factor
  - pickup(a₁), pickup(a₂), …, pickup(a₅₀₀)
- Thus forward-search can waste time trying lots of irrelevant actions
  - Need a good heuristic to guide the search
  - I’ll discuss one later
- But first, a very different kind of planning algorithm
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
The Planning Graph

- Search space for a relaxed version of the planning problem
- Alternating layers of ground literals and actions
  - At action-level $i$: all actions whose preconditions appear in state-level $i-1$
  - At state-level $i$: all the effects of all the actions at action-level $i$
  - Edges: preconditions and effects

**A maintenance action for a literal $l$.** It represents what happens if we don’t change $l$. 

state-level 0 (the literals true in $s_0$)
Example

- Due to Dan Weld (U. of Washington)

- Suppose you want to prepare dinner as a surprise for your sweetheart (who is asleep)
  \[ s_0 = \{\text{garbage, cleanHands, quiet}\} \]
  \[ g = \{\text{dinner, present, } \neg \text{garbage}\} \]

<table>
<thead>
<tr>
<th>Action</th>
<th>Preconditions</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>cook()</td>
<td>cleanHands</td>
<td>dinner</td>
</tr>
<tr>
<td>wrap()</td>
<td>quiet</td>
<td>present</td>
</tr>
<tr>
<td>carry()</td>
<td>none</td>
<td>(\neg\text{garbage, } \neg\text{cleanHands})</td>
</tr>
<tr>
<td>dolly()</td>
<td>none</td>
<td>(\neg\text{garbage, } \neg\text{quiet})</td>
</tr>
</tbody>
</table>

Also have the maintenance actions: one for each literal
Example (continued)

- state-level 0:
  \[ \{ \text{all atoms in } s_0 \} \cup \{ \text{negations of all atoms not in } s_0 \} \]

- action-level 1:
  \[ \{ \text{all actions whose preconditions are satisfied and non-mutex in } s_0 \} \]

- state-level 1:
  \[ \{ \text{all effects of all of the actions in action-level 1} \} \]

<table>
<thead>
<tr>
<th>Action</th>
<th>Preconditions</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>cook()</td>
<td>cleanHands</td>
<td>dinner</td>
</tr>
<tr>
<td>wrap()</td>
<td>quiet</td>
<td>present</td>
</tr>
<tr>
<td>carry()</td>
<td>none</td>
<td>\neg \text{garbage}, \neg \text{cleanHands}</td>
</tr>
<tr>
<td>dolly()</td>
<td>none</td>
<td>\neg \text{garbage}, \neg \text{quiet}</td>
</tr>
</tbody>
</table>

Also have the maintenance actions

- \neg \text{dinner}
- \neg \text{present}
Mutual Exclusion

- Two actions at the same action-level are mutex if
  - Inconsistent effects: an effect of one negates an effect of the other
  - Interference: one deletes a precondition of the other
  - 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
  - Inconsistent support: one is the negation of the other, or all ways of achieving them are pairwise mutex

Recursive propagation of mutexes
Example (continued)

- Augment the graph to indicate mutexes
- *carry* is mutex with the maintenance action for *garbage* (inconsistent effects)
- *dolly* is mutex with *wrap*
  - interference
- ~*quiet* is mutex with *present*
  - inconsistent support
- Each of *cook* and *wrap* is mutex with a maintenance operation

<table>
<thead>
<tr>
<th>Action</th>
<th>Preconditions</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>cook()</td>
<td>cleanHands</td>
<td>dinner</td>
</tr>
<tr>
<td>wrap()</td>
<td>quiet</td>
<td>present</td>
</tr>
<tr>
<td>carry()</td>
<td><em>none</em></td>
<td>~garbage, ~cleanHands</td>
</tr>
<tr>
<td>dolly()</td>
<td><em>none</em></td>
<td>~garbage, ~quiet</td>
</tr>
</tbody>
</table>

Also have the maintenance actions
Example (continued)

- Check to see whether there’s a possible solution
- Recall that the goal is
  - \{¬garbage, dinner, present\}
- Note that in state-level 1,
  - All of them are there
  - None are mutex with each other
- Thus, there’s a chance that a plan exists
- Try to find it
  - Solution extraction
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
        \( g' := \{ \text{the preconditions of} \)
        \( \text{the chosen actions} \} \)
    Solution-extraction(\( g' \), \( j-1 \))
end Solution-extraction
Example (continued)

- Two sets of actions for the goals at state-level 1
- Neither of them works
  - Both sets contain actions that are mutex

![Diagram showing state-level 0, action-level 1, and state-level 1 with actions like carry, dolly, cook, wrap, dinner, and present.](image-url)
Recall what the algorithm does

procedure Graphplan:
  for $k = 0, 1, 2, \ldots$
    $\Rightarrow$ *Graph expansion:*
      » create a “planning graph” that contains $k$ “levels”
    $\Rightarrow$ Check whether the planning graph satisfies a necessary (but insufficient) condition for plan existence
    $\Rightarrow$ 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 (continued)

- Go back and do more graph expansion

- Generate another action-level and another state-level
Solution extraction

Twelve combinations at level 4

- Three ways to achieve $\neg \text{garb}$
- Two ways to achieve $\text{dinner}$
- Two ways to achieve $\text{present}$
Several of the combinations look OK at level 2
Here’s one of them
Example (continued)

- Call Solution-Extraction recursively at level 2
- It succeeds
- Solution whose parallel length is 2
Earlier, I said

- Forward search can have a very large branching factor
  - pickup(a₁), pickup(a₂), …, pickup(a₅₀₀)
- Thus forward-search can waste time trying lots of irrelevant actions
  - Need a heuristic to guide the search

We can use planning graphs to compute such a heuristic
Getting Heuristic Values from a Planning Graph

Recall how GraphPlan works:

loop

Graph expansion: extend a “planning graph” forward from the initial state until we have achieved a necessary (but insufficient) condition for plan existence

this takes polynomial time

Solution extraction: search backward from the goal, looking for a correct plan if we find one, then return it

this takes exponential time

repeat
Using Planning Graphs to Compute $h(s)$

- In the graph, there are alternating layers of ground literals and actions.
- The number of “action” layers is a lower bound on the number of actions in the plan.
- Construct a planning graph, starting at $s$.
- $\Delta^g(s, g) =$ level of the first layer that “possibly achieves” the goal.
  - Some ways to improve this, but I’ll skip the details.
The FastForward Planner

- Use a heuristic function $h(s)$ similar to $\Delta^g(s, g)$
- Don’t want an A*-style search (takes too much memory)
- Instead, use a greedy procedure:

  until we have a solution, do
  expand the current state $s$
  $s :=$ the child of $s$ for which $h(s)$ is smallest
  (i.e., the child we think is closest to a solution)
The FastForward Planner

- Use a heuristic function $h(s)$ similar to $\Delta^g(s, g)$
- Don’t want an A*-style search (takes too much memory)
- Instead, use a greedy procedure:

  until we have a solution, do
  expand the current state $s$
  $s :=$ the child of $s$ for which $h(s)$ is smallest
  (i.e., the child we think is closest to a solution)
The FastForward Planner

- Use a heuristic function $h(s)$ similar to $\Delta^g(s,g)$
- Don’t want an A*-style search (takes too much memory)
- Instead, use a greedy procedure:

  until we have a solution, do
      expand the current state $s$
      $s :=$ the child of $s$ for which $h(s)$ is smallest
      (i.e., the child we think is closest to a solution)
The FastForward Planner

- Use a heuristic function $h(s)$ similar to $\Delta^g(s,g)$
- Don’t want an A*-style search (takes too much memory)
- Instead, use a greedy procedure:
  
  until we have a solution, do
  
  expand the current state $s$
  
  $s :=$ the child of $s$ for which $h(s)$ is smallest
  
  (i.e., the child we think is closest to a solution)
The FastForward Planner

- Use a heuristic function $h(s)$ similar to $\Delta g(s, g)$
- Don’t want an A*-style search (takes too much memory)
- Instead, use a greedy procedure:
  
  until we have a solution, do
  expand the current state $s$
  $s :=$ the child of $s$ for which $h(s)$ is smallest
  (i.e., the child we think is closest to a solution)

- Problem: can get caught in local minima
  - $h(s') > h(s)$ for every successor $s'$ of $s$
  - Escape by doing a breadth-first search until you find a node with lower cost

- Problem: can hit a dead end - in this case, FF fails
- No guarantee on whether FF will find a solution, or how good a solution
  - But FF works quite well on many classical planning problems
International Planning Competitions

  - Many of the planners in these competitions have incorporated ideas from GraphPlan and FastForward

- GraphPlan was developed in 1995
  - Several years before the competitions started

- FastForward was introduced in the 2000 International Planning Competition
  - It got an “outstanding performance” award
  - Large variance in how good its plans were, but it found them very quickly
Three Main Types of Planners

1. Domain-specific
2. Domain-independent
3. **Configurable**
   - Domain-independent planning engine
   - The input includes information about how to plan efficiently in a given problem domain

- I’ll now talk about a particular kind of configurable planner
Motivation

- For some planning problems, we may already have ideas about good ways to solve them
- Example: travel to a destination that’s far away:
  - Domain-independent planner:
    » many combinations vehicles and routes
  - Experienced human: small number of “recipes”
    e.g., flying:
    1. buy ticket from local airport to remote airport
    2. travel to local airport
    3. fly to remote airport
    4. travel to final destination
- How to get planning systems to use such recipes?
  - General approach: Hierarchical Task Network (HTN) planning
  - We’ll look at a simpler special case: Task-List Planning
Task-List Planning

- States and operators: same as in classical planning
- Instead of achieving a *goal*, we will want to accomplish a list of *tasks*
  - Recursively decompose tasks into smaller and smaller subtasks
  - At the bottom, actions that we know how to accomplish directly

- *Task*: an expression of the form \( t(u_1, \ldots, u_n) \)
  - \( t \) is a *task symbol*, and each \( u_i \) is a term

- Two kinds of task symbols (and tasks):
  - *primitive*: tasks that we know how to execute directly
    - task symbol is the head of an operator
  - *nonprimitive*: tasks that must be decomposed into subtasks
    - use *methods* (next slide)
**Methods**

- **Method**: a 4-tuple \( m = (\text{head, task, precond, subtasks}) \)
  - **head**: the method’s name, followed by list of variable symbols \((x_1, \ldots, x_n)\)
  - **task**: a nonprimitive task
  - **precond**: preconditions (literals)
  - **subtasks**: a sequence of tasks \( \langle t_1, \ldots, t_k \rangle \)

**air-travel**\((x, y, u, v)\)

- **task**: travel\((x, y)\)
- **precond**: far\((x, y)\), airport\((x, u)\), airport\((y, v)\)
- **subtasks**: get-ticket\((u, v)\), travel\((x, u)\), fly\((u, v)\), travel\((v, y)\)
Domains, Problems, Solutions

- Task-list planning domain: methods, operators
- Task-list planning problem: methods, operators, initial state, initial task list

- Solution: any executable plan that can be generated by recursively applying
  - methods to nonprimitive tasks
  - operators to primitive tasks
Example

Task: travel from UMD to UCLA
- Use air-travel method
- Use taxi-travel method for some of the subtasks
- The other subtasks (get-taxi, etc.) are primitive

Precond: far(UMD,UCLA), airport(UMD,BWI), airport(LAX,UCLA)

get-ticket (UMD,UCLA)  travel (UMD,BWI)  fly (BWI,LAX)  travel (LAX,UCLA)

taxi-travel(UMD,BWI)
Precond: ~far(UMD,BWI)

taxi-travel(LAX,UCLA)
Precond: ~far(LAX,UCLA)
Solving Task-List Planning Problems

- TFD(s,(t₁,…,tₖ))
  - if k=0 (i.e., no tasks) then return the empty plan
  - else if there is an action a such that head(a) = t₁ then
    - if s satisfies precond(a) then
      - return TFD(γ(s,t₁),(t₂,…,tₖ))
    - else return failure
  - else
    - A = {m : m is a method instance such that task(m)=t₁, and s satisfies precond(m)}
    - if active is empty then return failure
    - nondeterministically choose m in A
    - let u₁,…,uⱼ be m’s subtasks
    - return TFD(s, (u₁,…,uⱼ, t₂, …, tₖ))
Example

- $\text{TFD}(s,(t_1,\ldots,t_k))$
  - if $k=0$ (i.e., no tasks) then return the empty plan
  - else if there is an action $a$ such that $\text{head}(a) = t_1$ then
    » if $s$ satisfies $\text{precond}(a)$ then
      • return $\text{TFD}(\gamma(s,t_1),(t_2,\ldots,t_k))$
    » else return failure
  - else
    » $A = \{ m : m \text{ is a method instance such that} \quad \text{task}(m)=t_1, \text{ and } s \text{ satisfies } \text{precond}(m) \}$
    » if $\text{active}$ is empty then return failure
    » nondeterministically choose $m$ in $A$
    » let $u_1,\ldots,u_j$ be $m$’s subtasks
    » return $\text{TFD}(s,(u_1,\ldots,u_j, t_2, \ldots, t_k))$

---

Precond: far(x,y), airport(x,u), airport(y,v)
get-ticket $(u,v)$
travel $(x,u)$
fly $(u,v)$
travel $(v,y)$

Precond: ~far(x,y)
get-taxi
ride-taxi$(x,y)$
pay-driver

taxi-travel$(x,y)$

air-travel$(x,y,u,v)$

Task list:

$\langle \text{travel}(\text{UMD,UCLA}) \rangle$

Apply get-ticket method:

far(UMD,UCLA),
airport(UMD,BWI),
airport(UCLA,LAX)

Apply air-travel method:

$\langle \text{get-ticket}(\text{UMD,UCLA}) \rangle$
travel(UMD,BWI)
fly(BWI,LAX)
travel(LAX,UCLA)

Apply taxi-travel method:

$\langle \text{get-taxi} \rangle$
ride-taxi(UMD,BWI)
pay-driver
fly(BWI,LAX)
travel(LAX,UCLA)

$s_0: \quad \text{far(UMD,UCLA), airport(UMD,BWI), airport(UCLA,LAX)}$
Increasing Expressivity

- Easy to generalize this beyond classical planning
  - States can be arbitrary data structures

<table>
<thead>
<tr>
<th>Us: East declarer, West dummy</th>
<th>Opponents: defenders, South &amp; North</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contract: East – 3NT</td>
<td>On lead: West at trick 3</td>
</tr>
</tbody>
</table>

- Preconditions and effects can include
  - logical inferences (e.g., Horn clauses)
  - complex numeric computations
  - interactions with other software packages

- e.g., SHOP and SHOP2

http://www.cs.umd.edu/projects/shop
method travel-by-foot \((a, x, y)\)
  precond: \(distance(x, y) \leq 2\)
  task: travel\((a, x, y)\)
  subtasks: walk\((a, x, y)\)

method travel-by-taxi \((a, x, y)\)
  task: travel\((a, x, y)\)
  precond: \(cash(a) \geq 1.5 + 0.5 \times distance(x, y)\)
  subtasks: \(\langle\text{call-taxi}(a, x), \text{ride}(a, x, y), \text{pay-driver}(a, x, y)\rangle\)

operator walk\((a, x, y)\)
  precond: \(location(a) = x\)
  effects: \(location(a) \leftarrow y\)

operator call-taxi\((a, x)\)
  effects: \(location(taxi) \leftarrow x\)

operator ride-taxi\((a, x, y)\)
  precond: \(location(taxi) = x, location(a) = x\)
  effects: \(location(taxi) \leftarrow y, location(a) \leftarrow y\)

operator pay-driver\((a, x, y)\)
  precond: \(cash(a) \geq 1.5 + 0.5 \times distance(x, y)\)
  effects: \(cash(a) \leftarrow cash(a) - 1.5 + 0.5 \times distance(x, y)\)

Example

- Simple travel-planning domain
  - Go from one location to another
  - State = \{values of variables\}
Planning Problem: I am at home, I have $20, I want to go to a park 8 miles away

Initial task: travel(me,home,park)

Precondition: distance(home,park) ≤ 2

Precondition fails

Precondition succeeds

Decomposition into subtasks

Initial state $s_0 = \{\text{location}(me) = \text{home}, \text{cash}(me) = 20, \text{distance}(\text{home},\text{park}) = 8\}$

$\text{call-taxi}(me,\text{home})$

Precond: ...
Effects: ...

$\text{ride}(me,\text{home},\text{park})$

Precond: ...
Effects: ...

$\text{pay-driver}(me,\text{home},\text{park})$

Precond: ...
Effects: ...

Final state $s_3 = \{\text{location}(me) = \text{park}, \text{location}(\text{taxi}) = \text{park}, \text{cash}(me) = 14.50, \text{distance}(\text{home},\text{park}) = 8\}$
Comparison to Classical Planners

- **Advantages:**
  - Can encode “recipes” (standard ways do planning in a given domain) as collections of methods and operators
    - Helps the planning system do more-intelligent search - can speed up planning by many orders of magnitude (e.g., polynomial time versus exponential time)
    - Produces plans that correspond to how a human might solve the problem
  - Greater expressive power
    - Preconditions and effects can be computational algorithms

- **Disadvantages:**
  - More complicated than just writing classical operators
  - The author needs knowledge about planning in the given domain
SHOP2

- SHOP2:
  - Algorithm is a generalized version of TFD
  - Won an award in the AIPS-2002 Planning Competition
  - Freeware, open source
  - Downloaded more than 13,000 times
  - Used in hundreds (thousands?) of projects worldwide