Chapter 6
Planning-Graph Techniques
Motivation

- A big source of inefficiency in search algorithms is the *branching factor*
  - the number of children of each node
- e.g., a backward search may try lots of actions that can’t be reached from the initial state
- 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
Problem handled by GraphPlan*

- Pure STRIPS operators:
  - conjunctive preconditions
  - no negated preconditions
  - no conditional effects
  - no universal effects
- Finds “shortest parallel plan”
- Sound, complete and will terminate with failure if there is no plan.

*Version in [Blum& Furst IJCAI 95, AIJ 97]
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

- Alternating layers of ground literals and actions
  - All actions that might possibly occur at each time step
  - All of the literals asserted by those actions

*Maintenance* actions: propagate literals to the next level. These represent what happens if no action in the final plan affects the literal.
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
**Mutex**

- *Inconsistent Effects*
  - an effect of one negates an effect of the other

- E.g., `stack(a,b) & unstack(a,b)`

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>add handempty</td>
<td>delete handempty</td>
</tr>
<tr>
<td>(add ~handempty)</td>
<td></td>
</tr>
</tbody>
</table>
```

```
Inconsistent Effects
```
**Mutex**

- **Interference:**
  - one deletes a precondition of the other

- E.g., stack(a,b) & putdown(a)

  \[
  \downarrow \quad \downarrow
  \]

  deletes holding(a)  
  needs holding(a)
 Mutex

- **Competing needs:**
  - they have mutually exclusive preconditions
  - Their preconditions can’t be true at the same time
Mutex

- *Inconsistent support*:
  - one is the negation of the other
    E.g., handempty & ~handempty
  - or all ways of achieving them are pairwise mutex
    (from the previous reasons)
Valid plan

A valid plan is a planning graph where:

- Actions at the same level don’t interfere
- Each action’s preconditions are made true by the plan
- Goals are satisfied
GraphPlan algorithm

- Grow the planning graph (PG) until all goals are reachable and not mutex. (If PG levels off first, fail)
- Search the PG for a valid plan
- If non found, add a level to the PG and try again
Searching for a solution plan

- Backward chain on the planning graph
- Achieve goals level by level
- At level k, pick a subset of non-mutex actions to achieve current goals. Their preconditions become the goals for k-1 level.
- Build goal subset by picking each goal and choosing an action to add. Use one already selected if possible. Do forward checking on remaining goals (backtrack if can’t pick non-mutex action)
Solution Extraction

The set of goals we are trying to achieve

The level of the state $s_j$

A real action or a maintenance action

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
Plan Graph Search

If goals are present & non-mutex:
Choose action to achieve each goal
Add preconditions to next goal set
Example

due to Dan Weld (U. of Washington)

Suppose you want to prepare dinner as a surprise for your spouse (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:**
  \{ all atoms in \( s_0 \) \} \( \cup \)
  \{ negations of all atoms not in \( s_0 \) \}

- **action-level 1:**
  \{ all actions whose preconditions are satisfied in \( s_0 \) \}

- **state-level 1:**
  \{ 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 )garbage, ( \neg )cleanHands</td>
</tr>
<tr>
<td>dolly()</td>
<td>none</td>
<td>( \neg )garbage, ( \neg )quiet</td>
</tr>
</tbody>
</table>

Also have the maintenance actions

\( \neg \)dinner \hspace{1cm} \neg \)dinner
\( \neg \)present \hspace{1cm} \neg \)present
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>none</td>
<td>¬garbage, ¬cleanHands</td>
</tr>
<tr>
<td>dolly()</td>
<td>none</td>
<td>¬garbage, ¬quiet</td>
</tr>
</tbody>
</table>

Also have the maintenance actions
Example (continued)

- Check to see whether there’s a possible plan
- Recall that the goal is
  - \( \{\neg \text{garbage}, \text{dinner}, \text{present}\} \)
- Note that
  - All are possible in \( s_1 \)
  - None are mutex with each other
- Thus, there’s a chance that a plan exists
- Try to find it
  - Solution extraction
Example (continued)

- Two sets of actions for the goals at state-level 1
- Neither works: both sets contain actions that are mutex
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
Observation 1

Propositions monotonically increase
(always carried forward by no-ops)
Observation 2

Actions monotonically increase
Observation 3

Proposition mutex relationships monotonically decrease
Observation 4

Action mutex relationships monotonically decrease
Observation 5

Planning Graph ‘levels off’.

- After some time $k$ all levels are identical
- Because it’s a finite space, the set of literals never decreases and mutexes don’t reappear.
Termination for unsolvable problems

- Graphplan records (memoizes) sets of unsolvable goals:
  - $U(i, t) =$ unsolvable goals at level $i$ after stage $t$.
- More efficient: early backtracking
- Also provides necessary and sufficient conditions for termination:
  - Assume plan graph levels off at level $n$, stage $t > n$
  - If $U(n, t-1) = U(n, t)$ then we know we’re in a loop and can terminate safely.
Expressive Languages

- Negated preconditions
- Disjunctive preconditions
- Universally quantified preconditions, effects
- Conditional effects
- Iterative/Recursive Actions
Comparison with PSP

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

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

- Can alleviate (but not eliminate) this problem by assigning data types to the variables and constants
  - Only instantiate variables to terms of the same data type
History

- Before Graphplan came out, most planning researchers were working on PSP-like planners
  - POP, SNLP, UCPOP, etc.
- Graphplan caused a sensation because it was so much faster
- Many subsequent planning systems have used ideas from it
  - IPP, STAN, GraphHTN, SGP, Blackbox, TGP, LPG
- Many of them are much faster than the original Graphplan