

Lecture slides for
Automated Planning: Theory and Practice

Chapter 6

Planning-Graph Techniques

Dana S. Nau

University of Maryland

Fall 2009

History

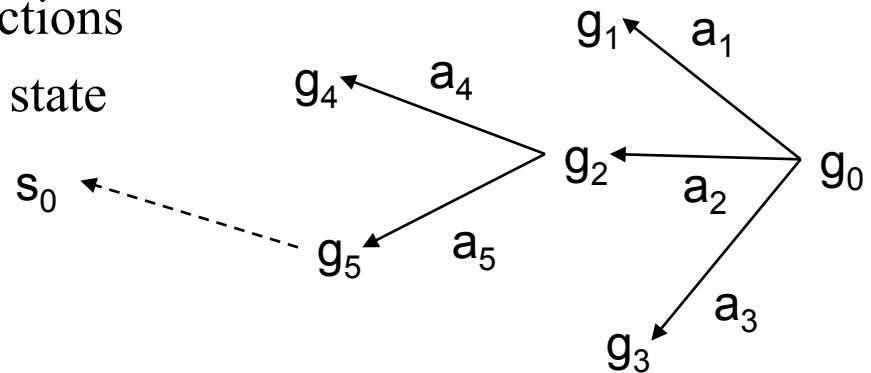
- Before Graphplan came out, most planning researchers were working on PSP-like planners
 - ◆ POP, SNLP, UCPOP, etc.
- Graphplan caused a sensation because it was so much faster
- Many subsequent planning systems have used ideas from it
 - ◆ IPP, STAN, GraphHTN, SGP, Blackbox, Medic, TGP, LPG
 - ◆ Many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

Outline

- Motivation
- The Graphplan algorithm
- Constructing planning graphs
 - ◆ example
- Mutual exclusion
 - ◆ example (continued)
- Doing solution extraction
 - ◆ example (continued)
- Discussion

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

Graphplan

procedure Graphplan:

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

- ◆ *Graph expansion:*

- » create a “planning graph” that contains k “levels”

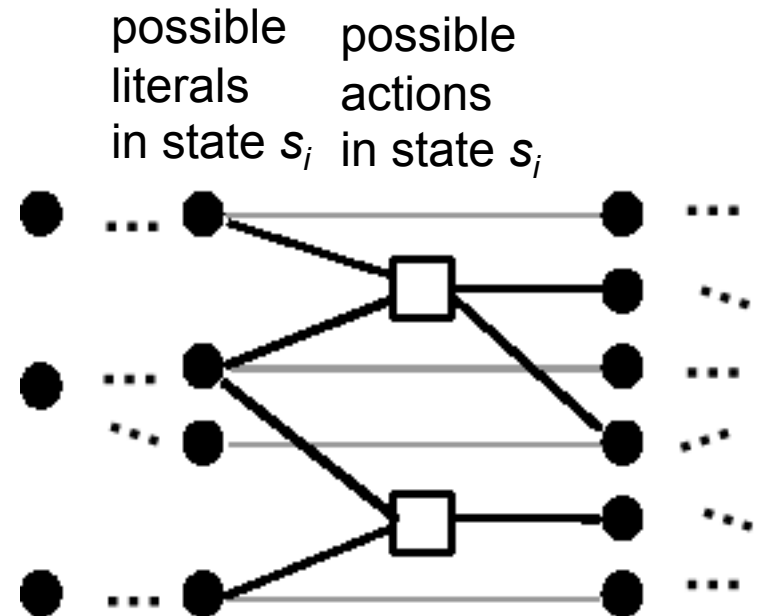
- ◆ Check whether the planning graph satisfies a necessary (but insufficient) condition for plan existence

relaxed
problem

- ◆ If it does, then

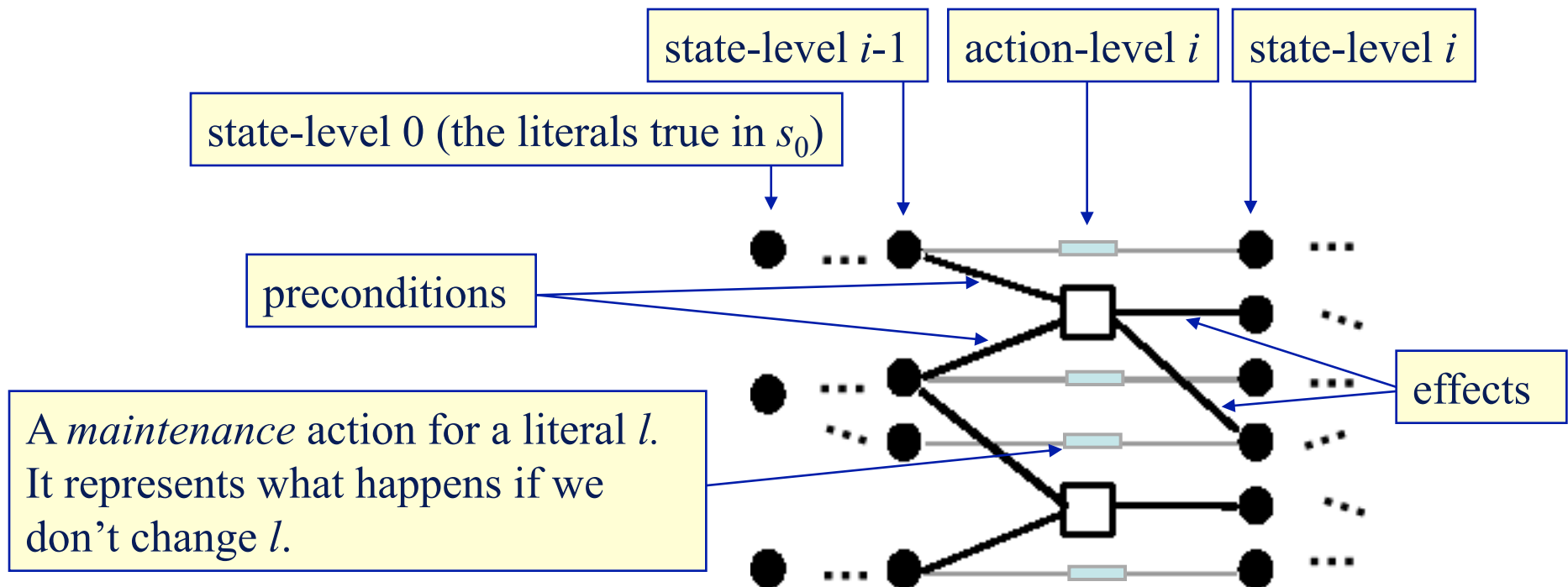
- » do *solution extraction:*

- 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
 - ◆ 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



Example

- Due to Dan Weld (U. of Washington)
- Suppose you want to prepare dinner as a surprise for your sweetheart (who is asleep)

$s_0 = \{\text{garbage, cleanHands, quiet}\}$

$g = \{\text{dinner, present, } \neg\text{garbage}\}$

<u>Action</u>	<u>Preconditions</u>	<u>Effects</u>
cook()	cleanHands	dinner
wrap()	quiet	present
carry()	<i>none</i>	\neg garbage, \neg cleanHands
dolly()	<i>none</i>	\neg garbage, \neg quiet

Also have the maintenance actions: one for each literal

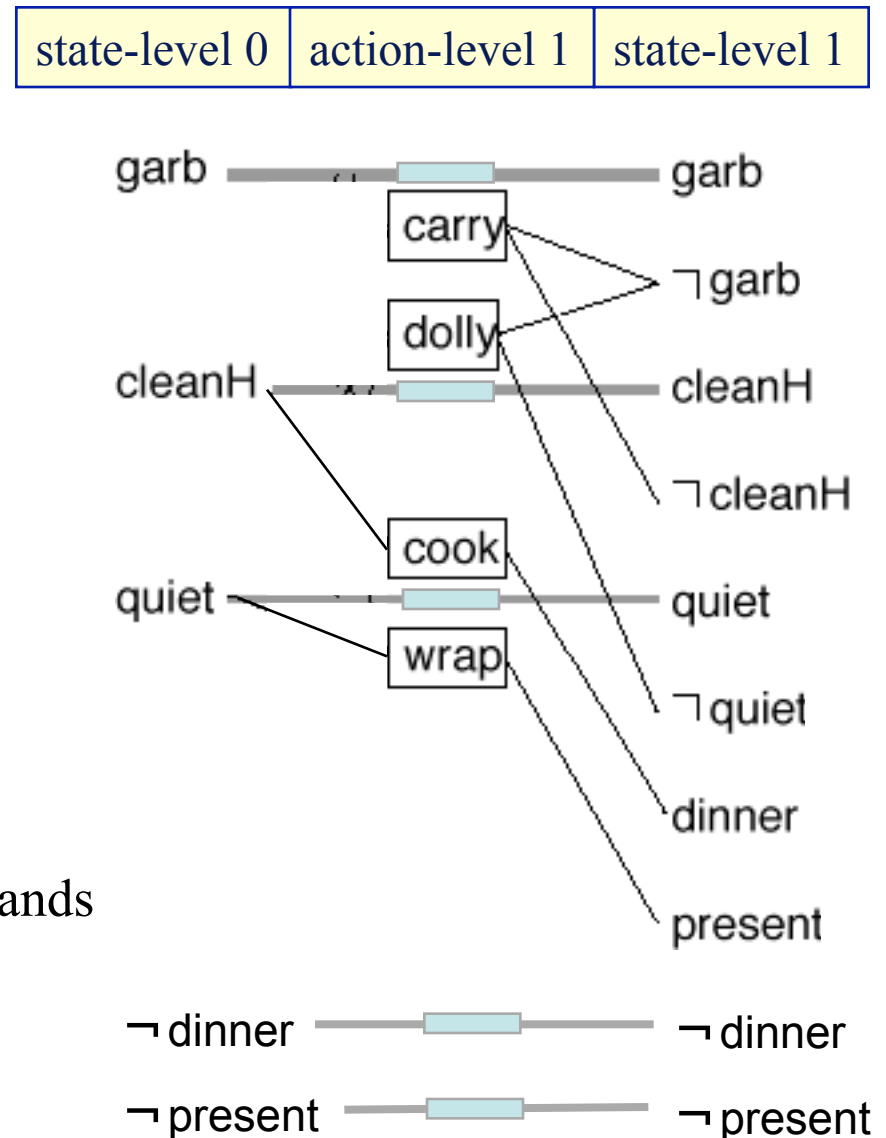
Example (continued)

- 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 and non-mutex in s_0 }
- state-level 1:
 - {all effects of all of the actions in action-level 1}

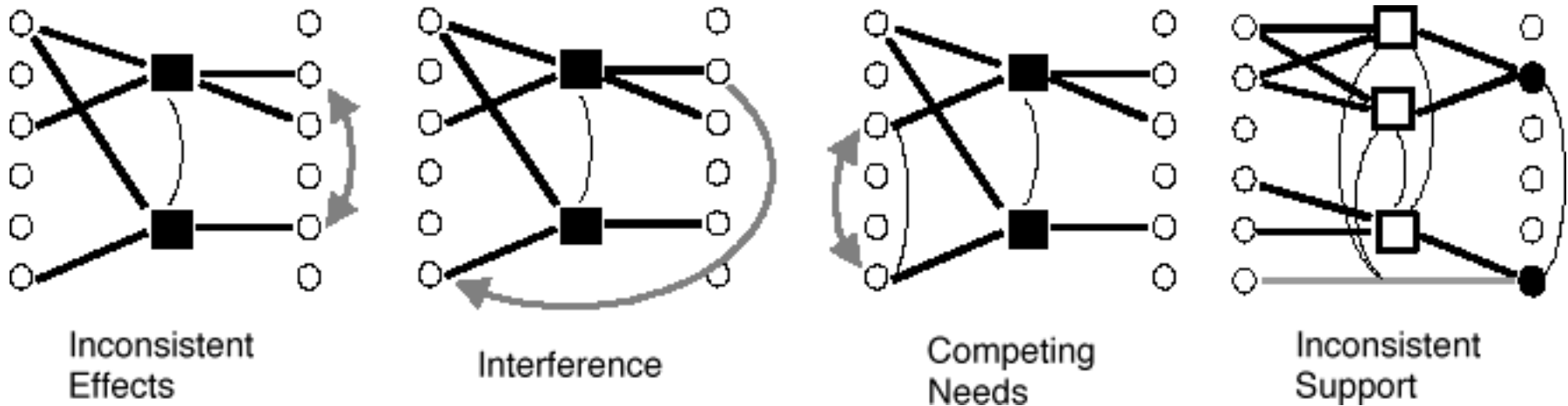
Action Preconditions Effects

cook()	cleanHands	dinner
wrap()	quiet	present
carry()	<i>none</i>	\neg garbage, \neg cleanHands
dolly()	<i>none</i>	\neg garbage, \neg quiet

Also have the maintenance actions



Mutual Exclusion

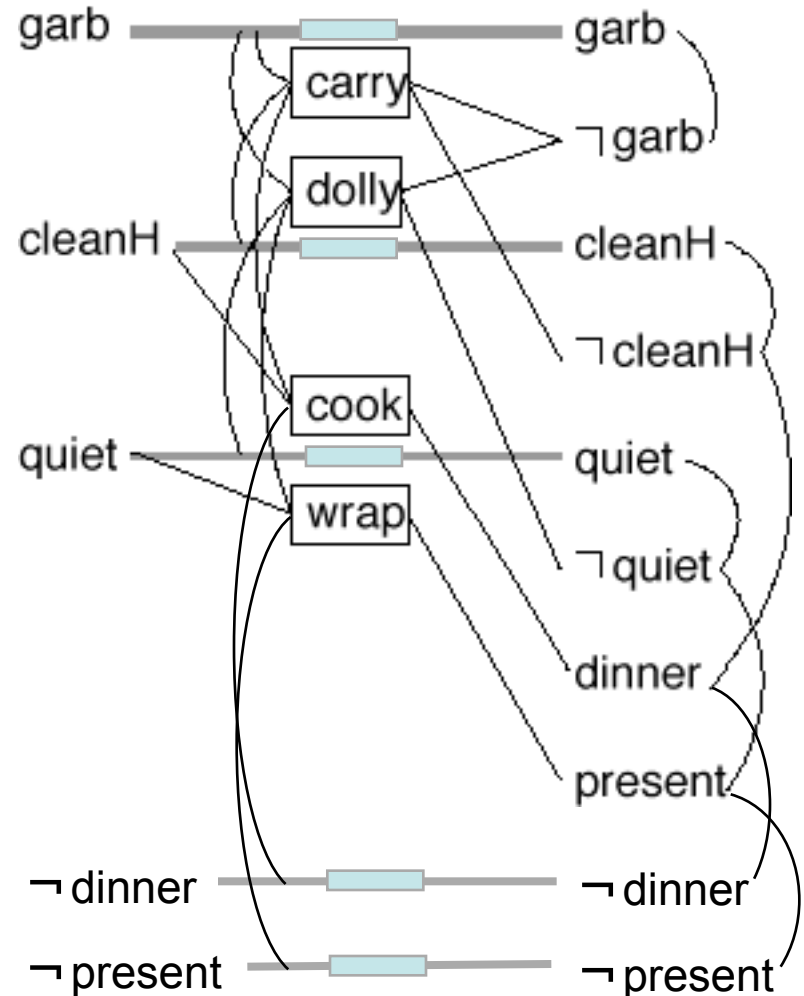


- Two actions at the same action-level are mutex if
 - ◆ *Inconsistent effects*: an effect of one negates an effect of the other
 - ◆ *Interference*: one deletes a precondition of the other
 - ◆ *Competing needs*: **they have mutually exclusive preconditions**
- Otherwise they don't interfere with each 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
- \sim *quiet* is mutex with *present*
 - ◆ inconsistent support
- each of *cook* and *wrap* is mutex with a maintenance operation

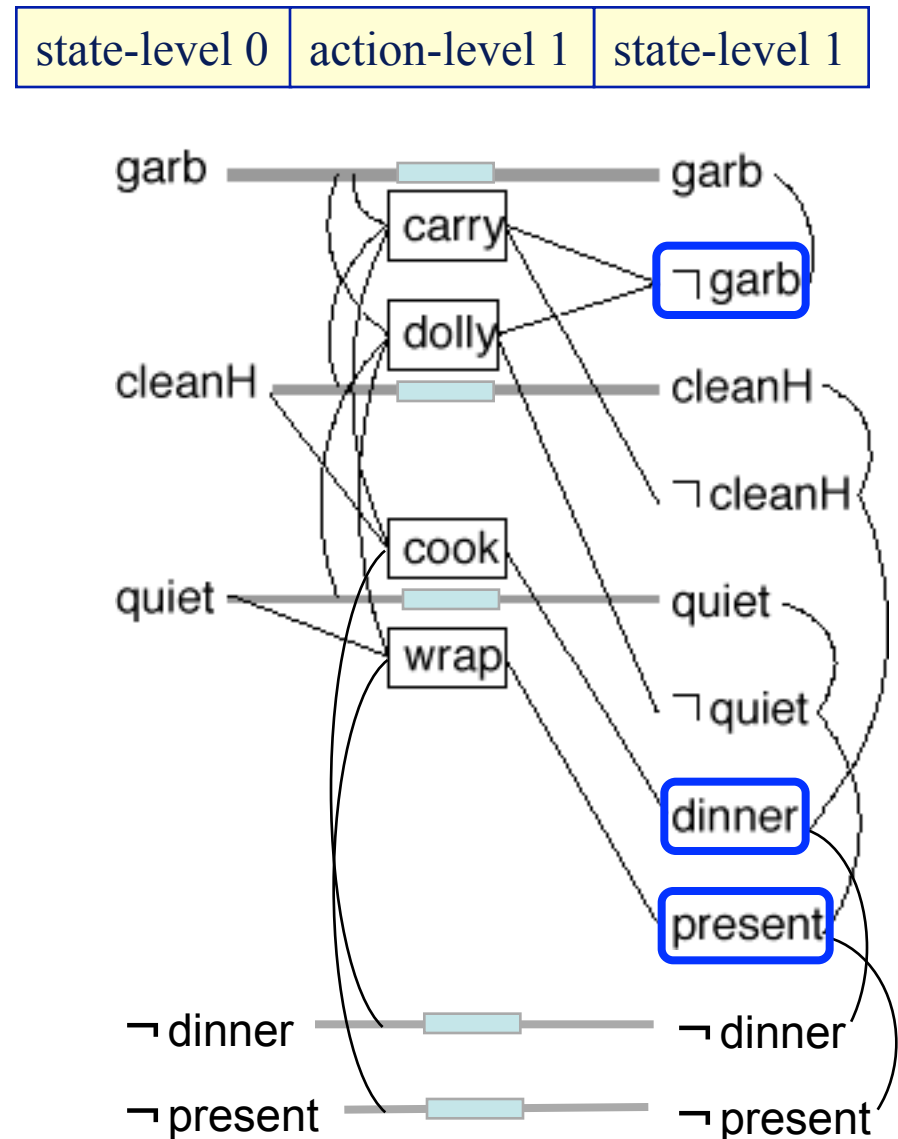


<u>Action</u>	<u>Preconditions</u>	<u>Effects</u>
<i>cook</i> ()	<i>cleanHands</i>	<i>dinner</i>
<i>wrap</i> ()	<i>quiet</i> <i>present</i>	
<i>carry</i> ()	<i>none</i> \neg <i>garbage</i> , \neg <i>cleanHands</i>	
<i>dolly</i> ()	<i>none</i> \neg <i>garbage</i> , \neg <i>quiet</i>	

Also have the maintenance actions

Example (continued)

- Check to see whether there's a possible solution
- Recall that the goal is
 - ◆ $\{\neg \textit{garbage}, \textit{dinner}, \textit{present}\}$
- 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



Solution Extraction

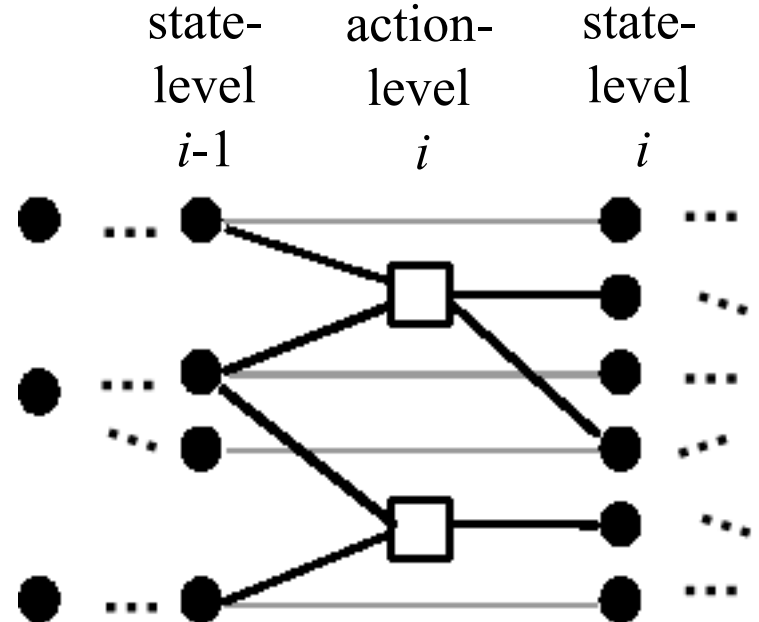
The set of goals we are trying to achieve

The level of the state s_j

```

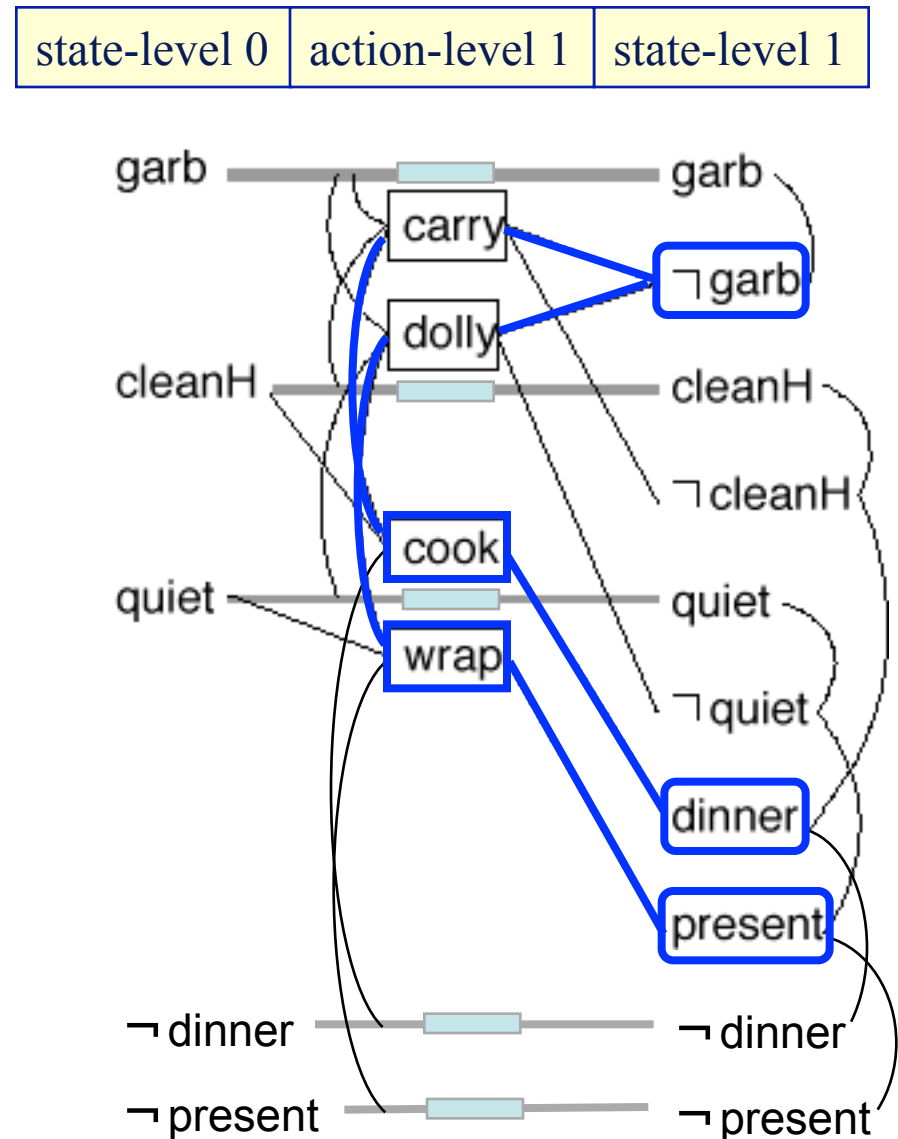
procedure Solution-extraction( $g, j$ )
  if  $j=0$  then return the solution
  for each literal  $l$  in  $g$ 
    nondeterministically choose an action
      to use in state  $s_{j-1}$  to achieve  $l$ 
  if any pair of chosen actions are mutex
    then backtrack
   $g' := \{$ the preconditions of
    the chosen actions $\}$ 
  Solution-extraction( $g', j-1$ )
end Solution-extraction
  
```

A real action or a maintenance action



Example (continued)

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



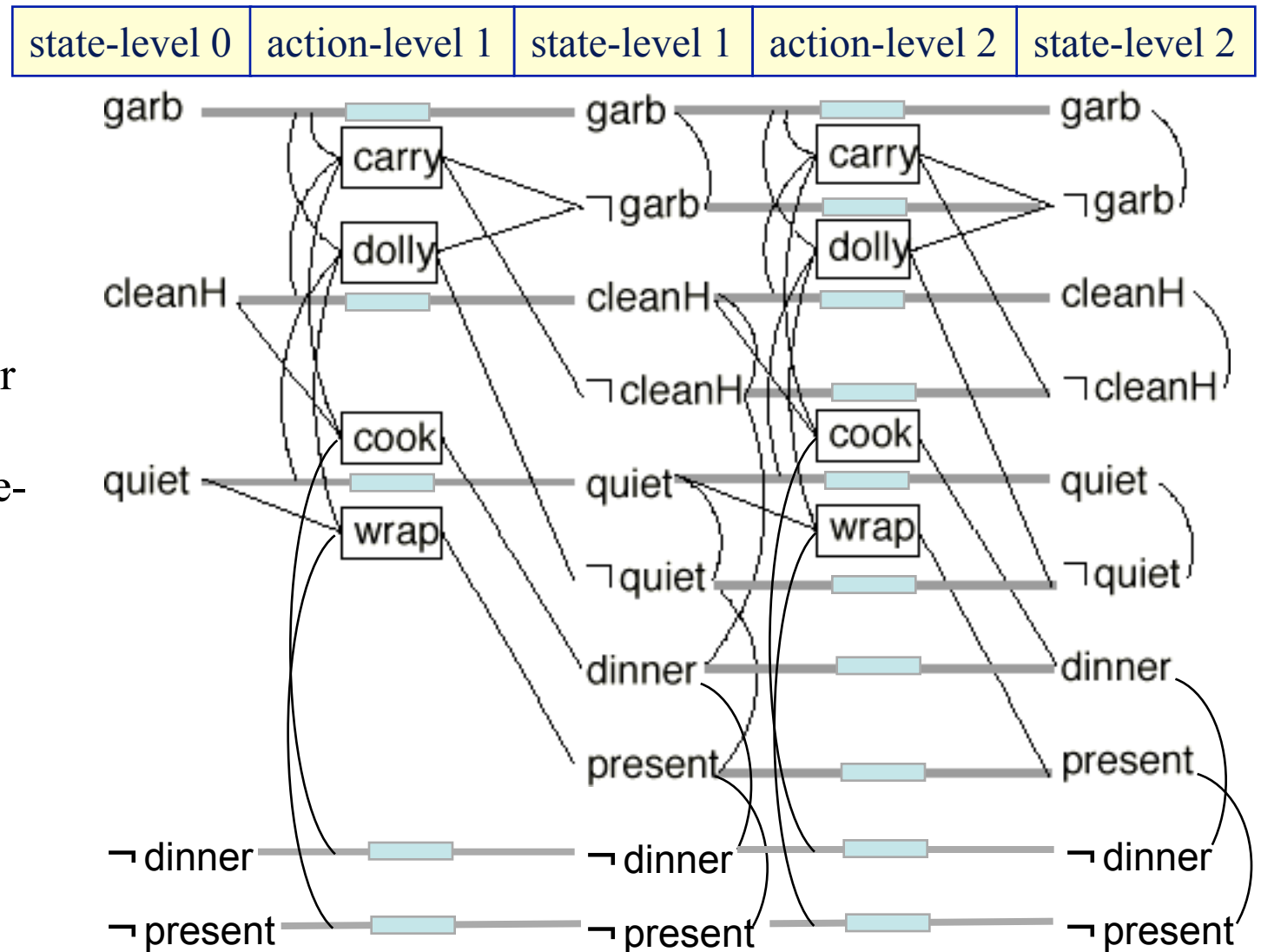
Recall what the algorithm does

procedure Graphplan:

- for $k = 0, 1, 2, \dots$
 - ◆ *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

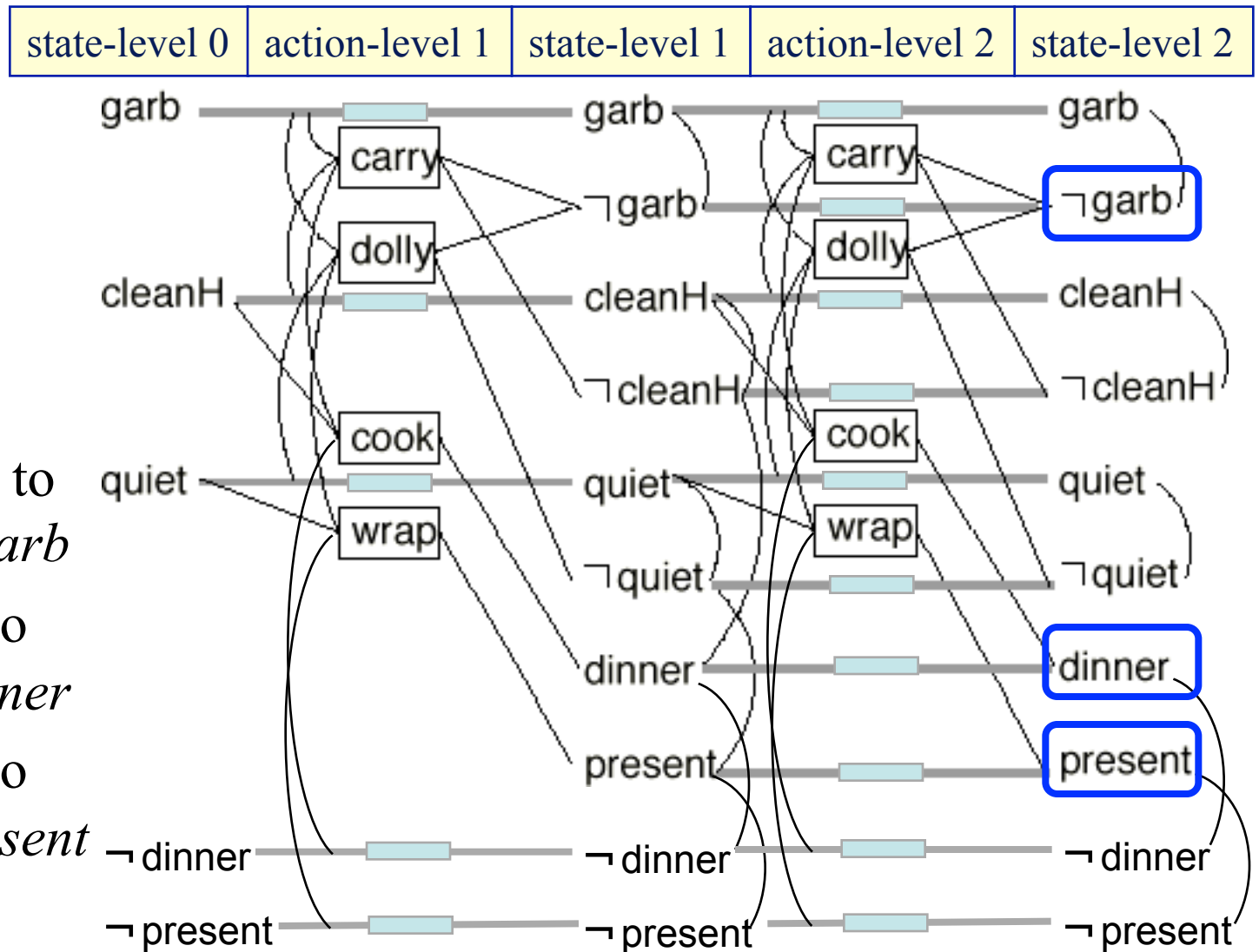
Example (continued)

- Go back and do more graph expansion
- Generate another action-level and another state-level



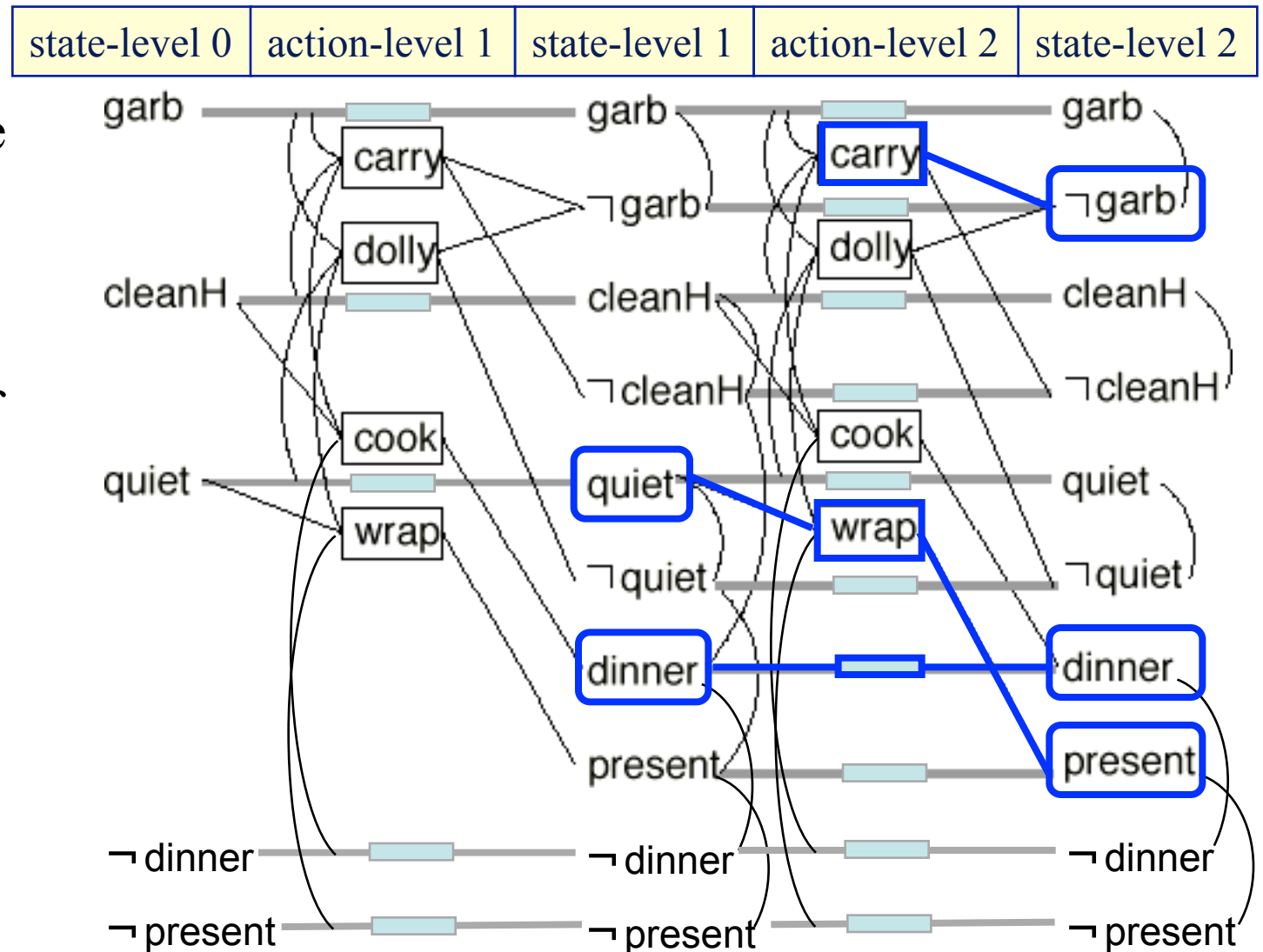
Example (continued)

- Solution extraction
- Twelve combinations at level 4
 - ◆ Three ways to achieve $\neg garb$
 - ◆ Two ways to achieve *dinner*
 - ◆ Two ways to achieve *present*



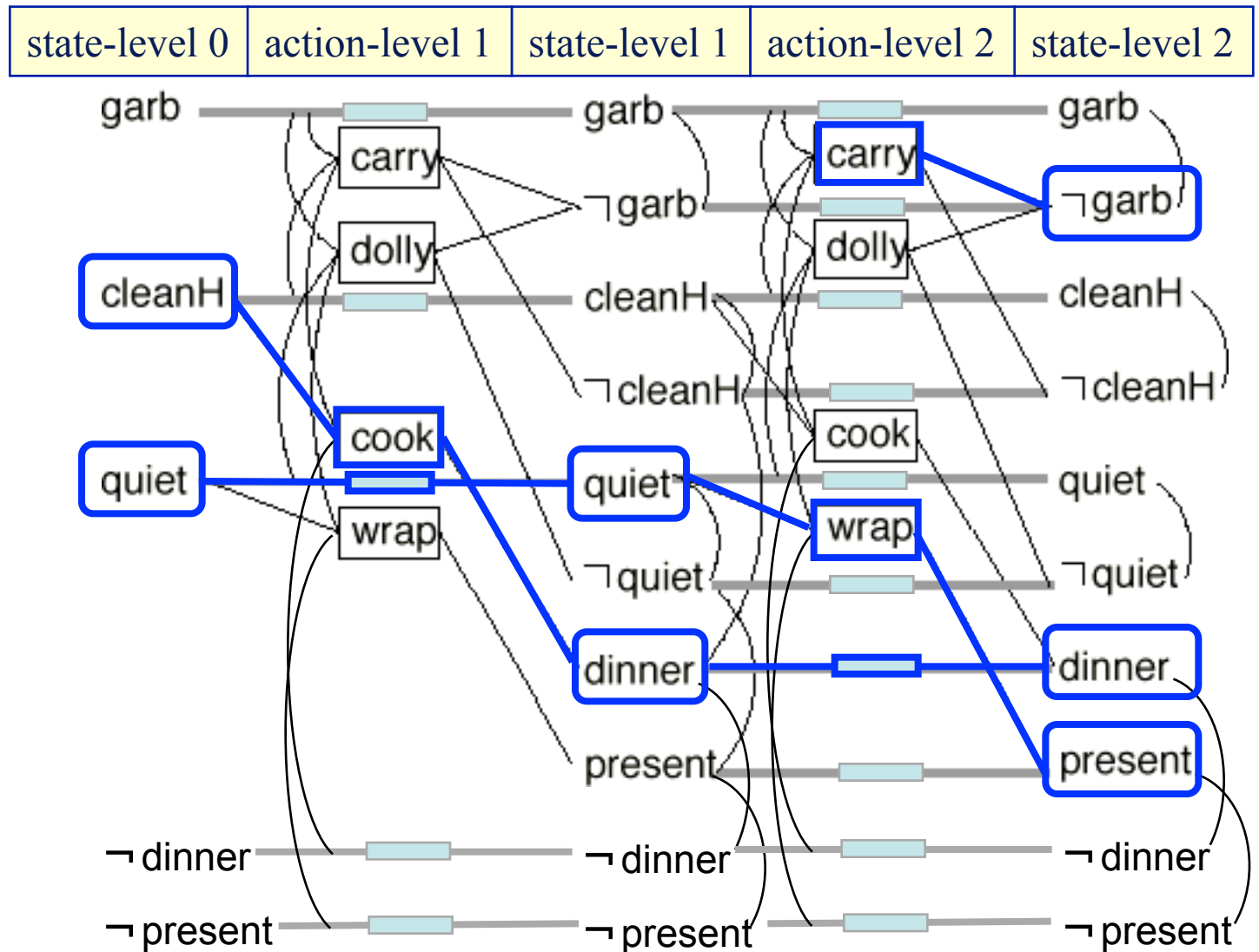
Example (continued)

- 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



Comparison with Plan-Space Planning

- 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
- For classical planning, the advantage outweighs the disadvantage
 - ◆ GraphPlan solves classical planning problems much faster than PSP