# Graphplan

Jiří Vokřínek

A4M36PAH - 15.4.2012

# Materials

- Steven M. LaValle: *Planning Algorithms*, 2006
  http://planning.cs.uiuc.edu/
- Malik Ghallab, Dana Nau, Paolo Traverso: *Automated Planning: Theory and Practice*, 2004
  http://projects.laas.fr/planning/

- Dana Nau's lecture slides
  http://www.cs.umd.edu/~nau/planning/slides/chapter06.pdf
- Gerhard Wickler's lecture slides
  http://www.inf.ed.ac.uk/teaching/courses/plan/slides/Graphplan-Slides.pdf

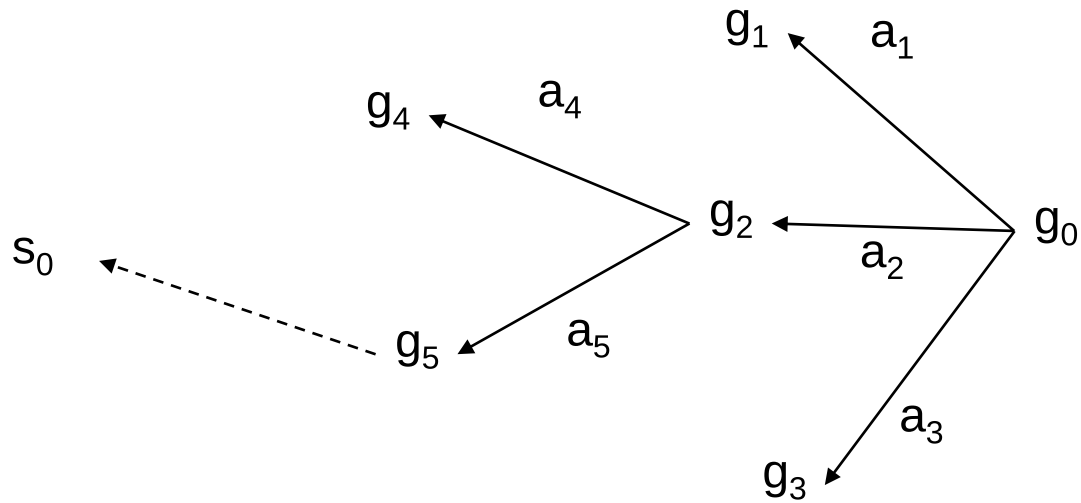
SOME RIGHTS RESERVED

# 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

# 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



# Motivation

- 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

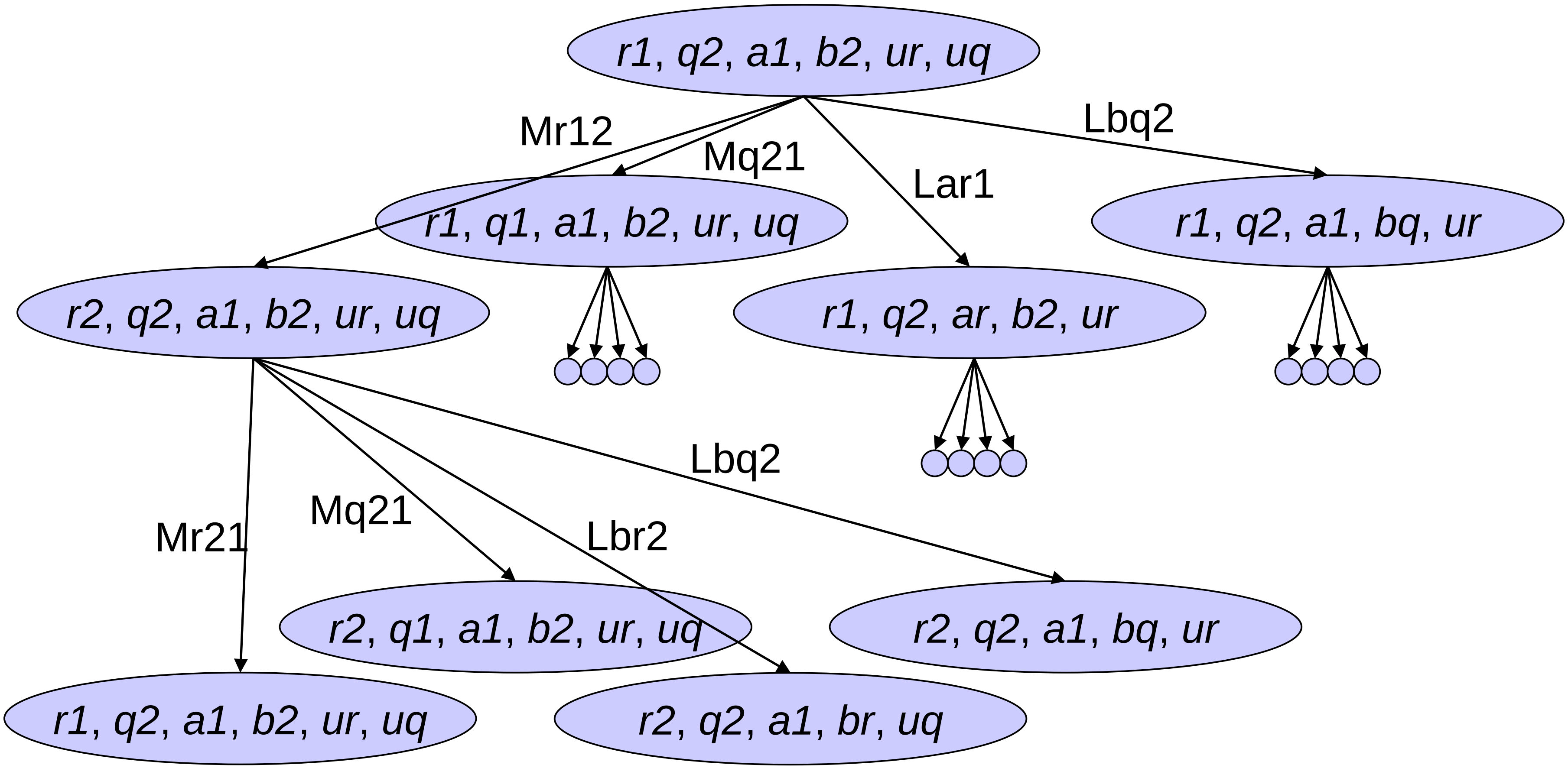
# Reachability Tree

- Tree structure, where:
  - Nodes are states
  - Edges correspond to actions
  - Root is initial state $s_0$
  - Children of node *s* are $\Gamma(s)$

- All nodes in reachability tree are $\Gamma^{>}(s_0)$
  - All nodes to depth *d* are $\Gamma^{d}(s_0)$
  - Solves problems with up to *d* actions in solution

- Problem: $O(k^d)$ nodes;
  *k* = applicable actions per state

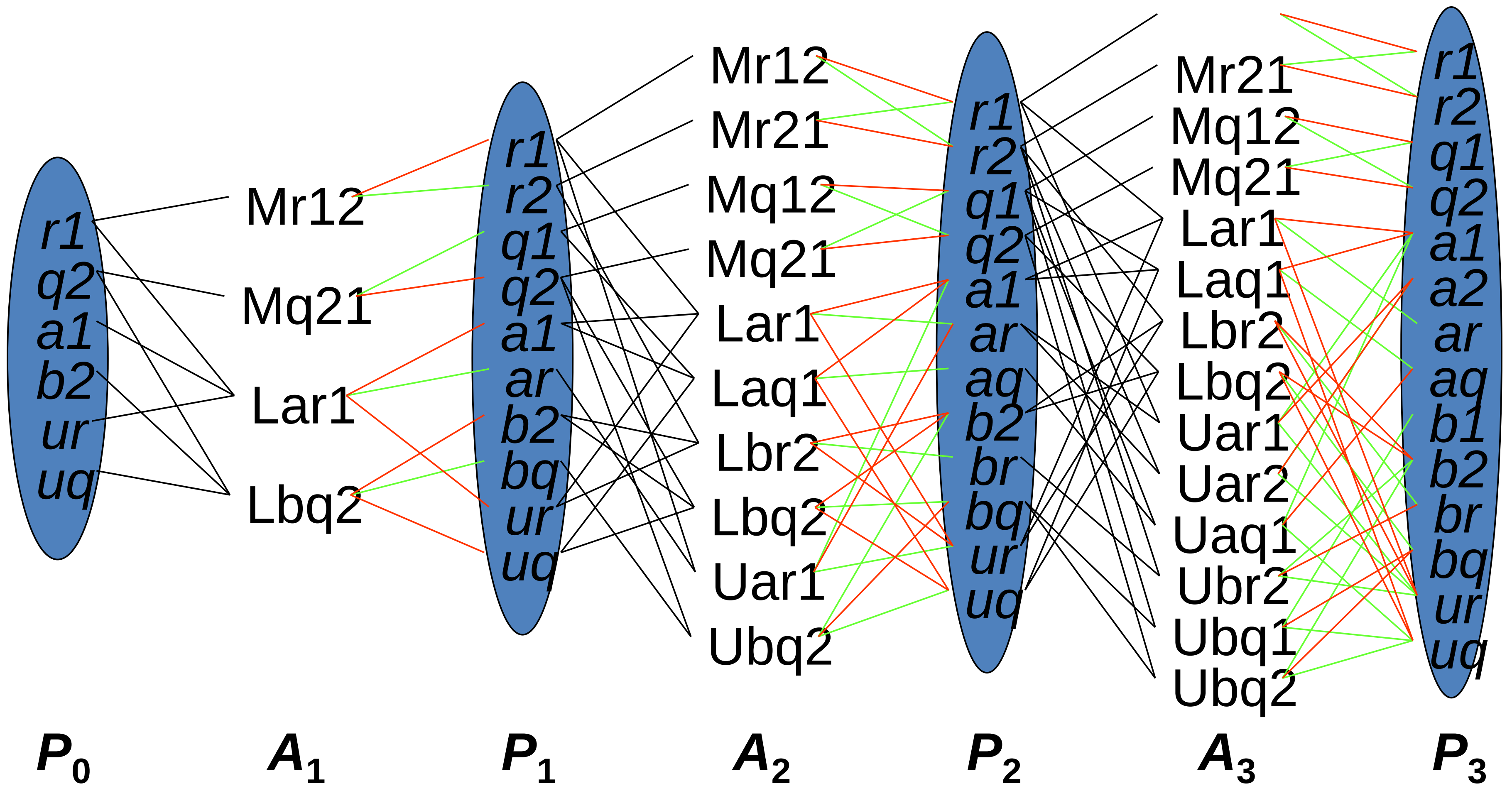
# Reachability Tree



# Reachability with Planning Graph

- Layered directed graph $G=(N,E)$:
  - Nodes - $P_0 \cup A_1 \cup P_1 \cup A_2 \cup P_2 \cup \ldots$
    - state proposition layers: $P_0$, $P_1$, …
    - action layers: $A_1$, $A_2$, …
  - Edges
    - from proposition $p \in P_{j-1}$ to action $a \in A_j$:
    - from action $a \in A_j$ to layer $p \in P_j$:

# Reachability with Planning Graph



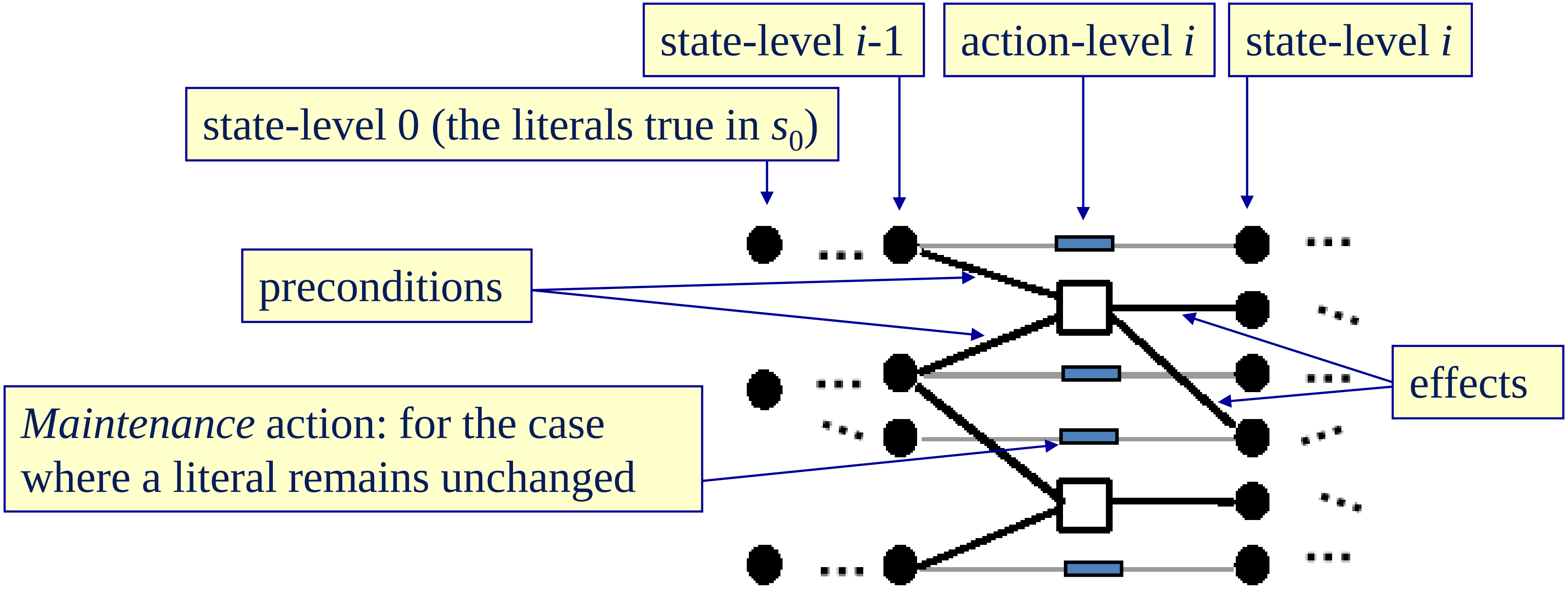
# Reachability with Planning Graph

- Reachability analysis:
  - if a goal $g$ is reachable from initial state $s_0$
  - then there will be a proposition layer $P_g$ in the planning graph such that $g \subseteq P_g$

- Necessary condition, but not sufficient
- Low complexity:
  - planning graph is of polynomial size and
  - can be computed in polynomial time

# The Planning Graph

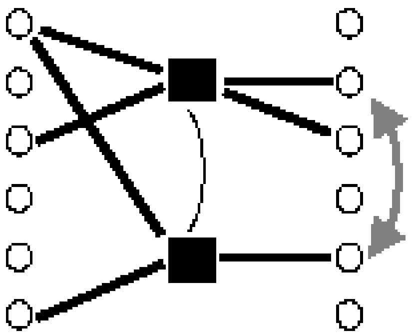
- Search space for a relaxed version of the planning problem
- Alternating layers of ground literals and actions
  - Nodes at action-level *i*: actions that might be possible to execute at time *i*
  - Nodes at state-level *i:* literals that might possibly be true at time *i*
  - Edges: preconditions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



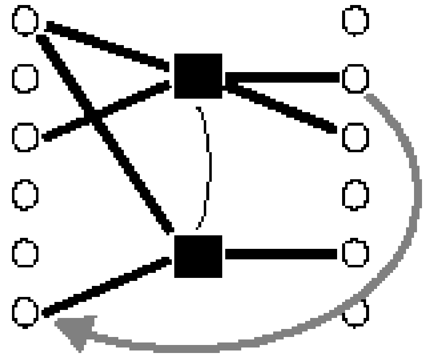
# Planning Graph Construction

- The planning graph is constructed **layer by layer**
- Every positive literal in $s_0$ is placed into state-level 0, along with the negation of every positive literal not in $s_0$
- Every *i-th* action-level contains all operators for which their preconditions are a subset of state-level *i-1*
- For each possible literal *l* a **trivial operator** is constructed for which *l* is the only precondition and effect in every action-level
- Every *i-th* state-level is the union of the effect of operators of action-level *i*
- For every level, maintain conflicts (**mutex condition**)
- The iterations continue until the planning **graph stabilizes**, i.e. both action-level and state-level in *i+1* is the same as in *i-th* iteration

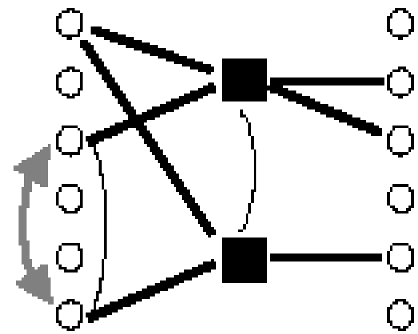
# Mutex Condition



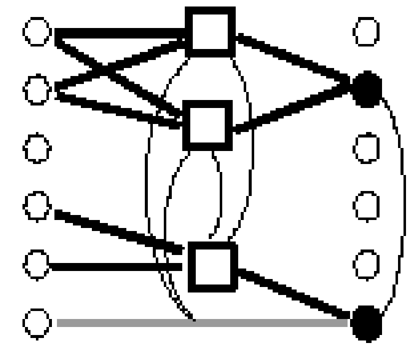
Inconsistent Effects



Interference



Competing Needs



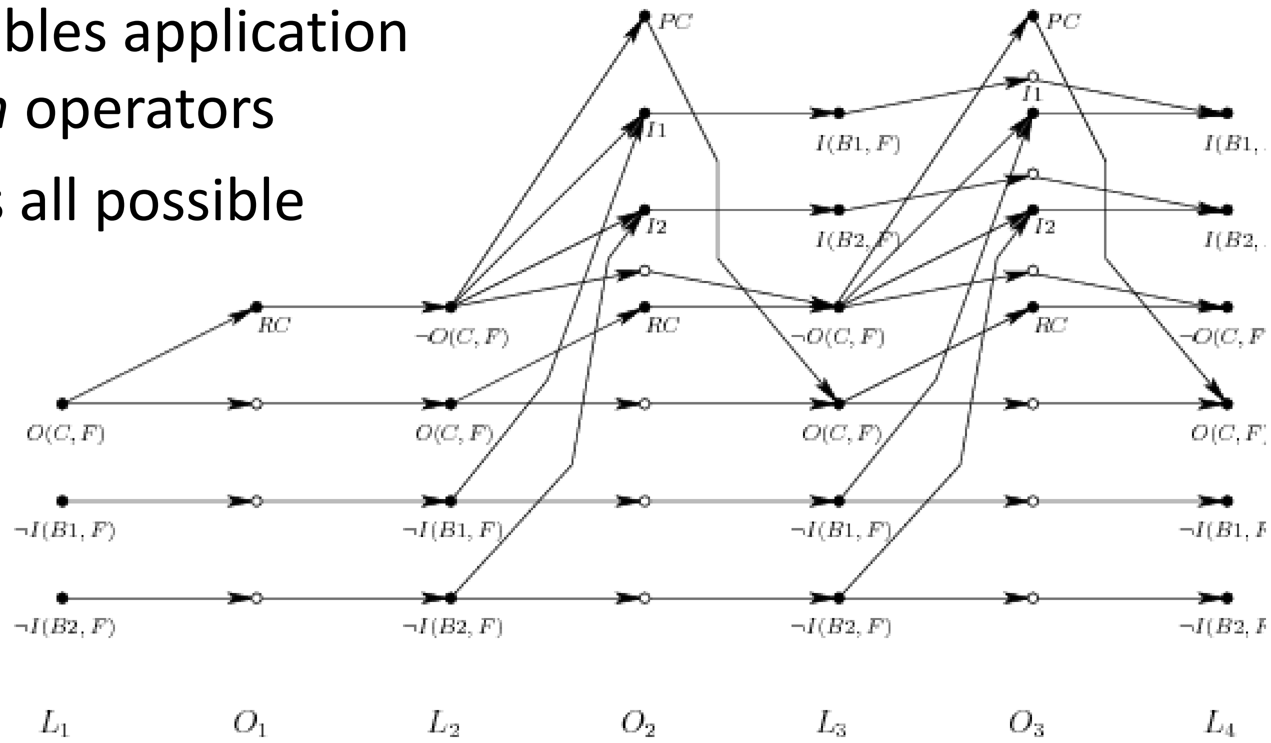
Inconsistent Support

- 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

# Graph Stabilization

- Flashlight example
  - $L_1$ expenses initial state
  - $O_1$ contains *RemoveCap* operator and three trivial operators
  - *-O(C,F)* enables application of *Insertion* operators
  - $O_3$ contains all possible operators
  - $L_3$=$L_4$
  - $O_3$=$O_4$



# Graphplan

Procedure Graphplan:

- for *k* = 0, 1, 2, …
  - *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

  hard part

  - If the graph is stabilized, solution is unreachable

# 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*

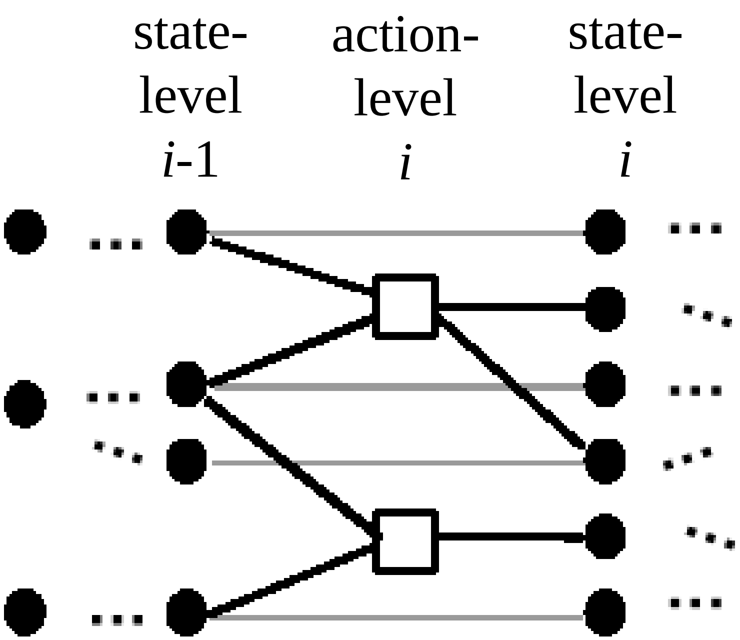
if any pair of chosen actions are mutex

then backtrack

*g'* := {the preconditions of the chosen actions}

Solution-extraction(*g'*, *j*–1)

end Solution-extraction



# Example

- Suppose you want to prepare dinner as a surprise for your sweetheart (who is asleep)

  $s_0$ = {garbage, cleanHands, quiet}

  $g$ = {dinner, present, ¬garbage}

| Action | Preconditions | Effects |
| --- | --- | --- |
| cook() | cleanHands | dinner |
| wrap() | quiet | present |
| carry() | *none* | ¬garbage, ¬cleanHands |
| dolly() | *none* | ¬garbage, ¬quiet |

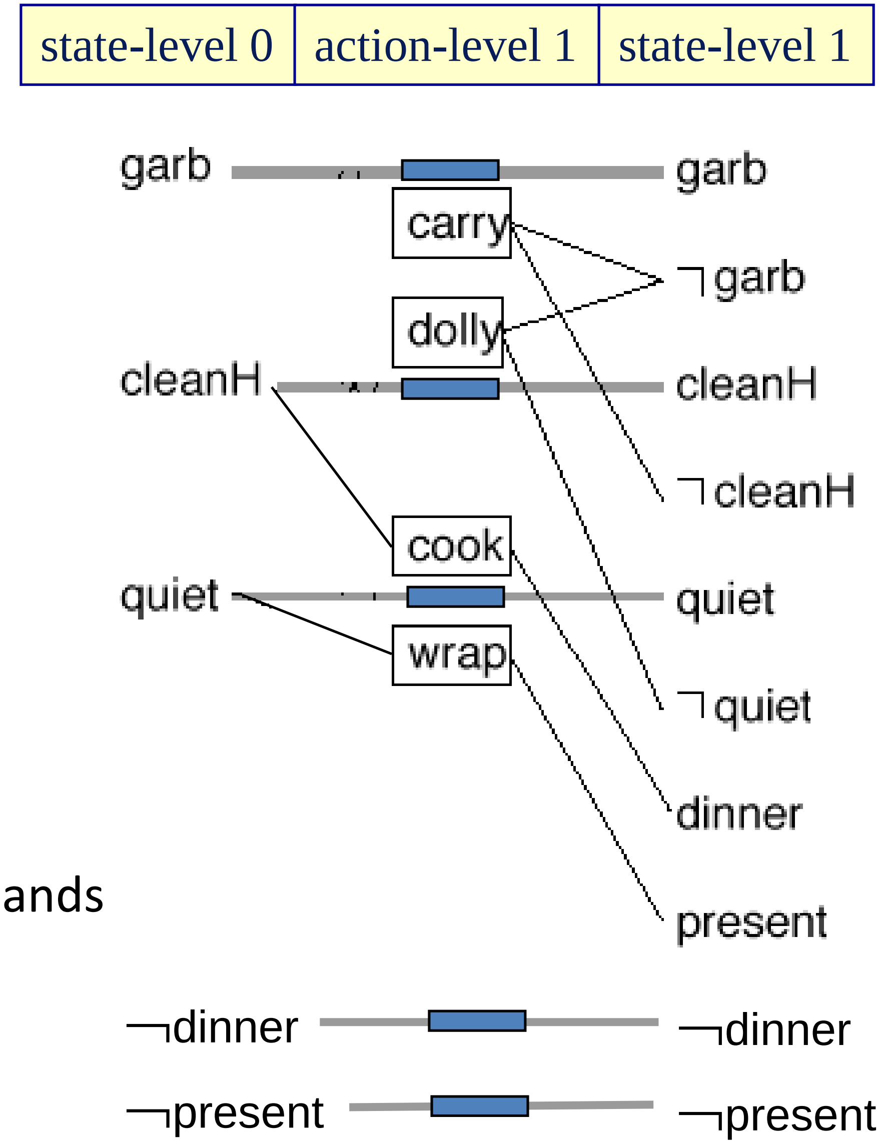
Also have the maintenance actions: one for each literal

# Example (continued)

- state-level 0:
  {all atoms in $s_0$} U
  {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}

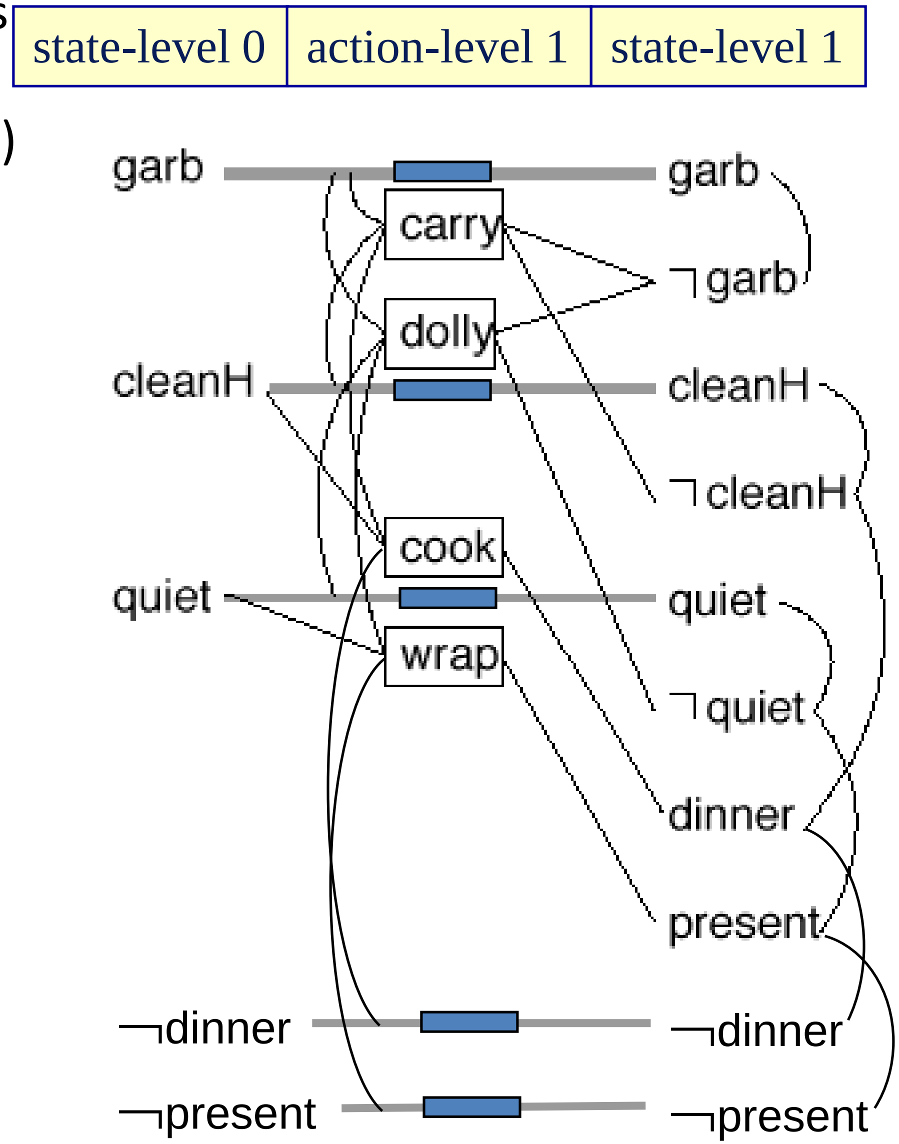
| Action | Preconditions | Effects |
| --- | --- | --- |
| cook() | cleanHands | dinner |
| wrap() | quiet | present |
| carry() | *none* | ¬garbage, ¬cleanHands |
| dolly() | *none* | ¬garbage, ¬quiet |

Also have the maintenance actions



# 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

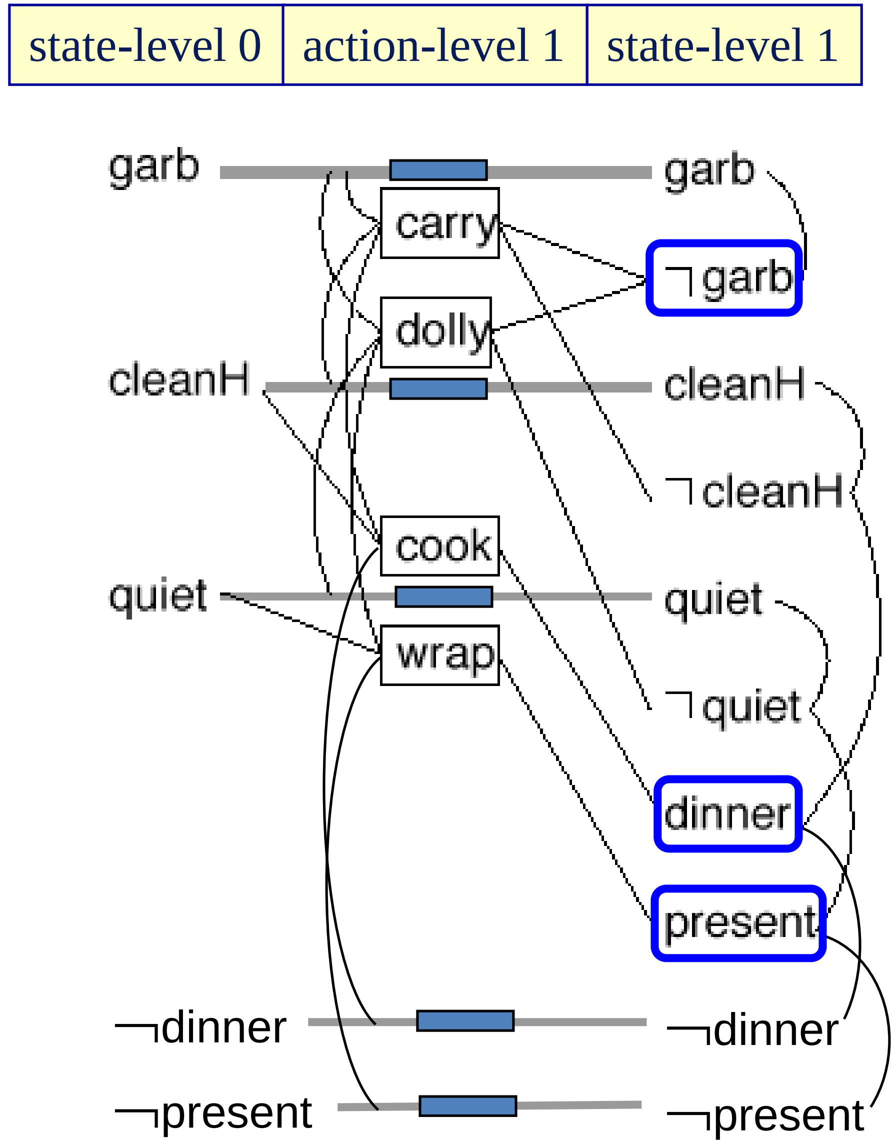



| **Action** | **Preconditions** | **Effects** |
|---|---|---|
| cook() | cleanHands | dinner |
| wrap() | quiet | present |
| carry() | *none* | ¬garbage, ¬cleanHands |
| dolly() | *none* | ¬garbage, ¬quiet |

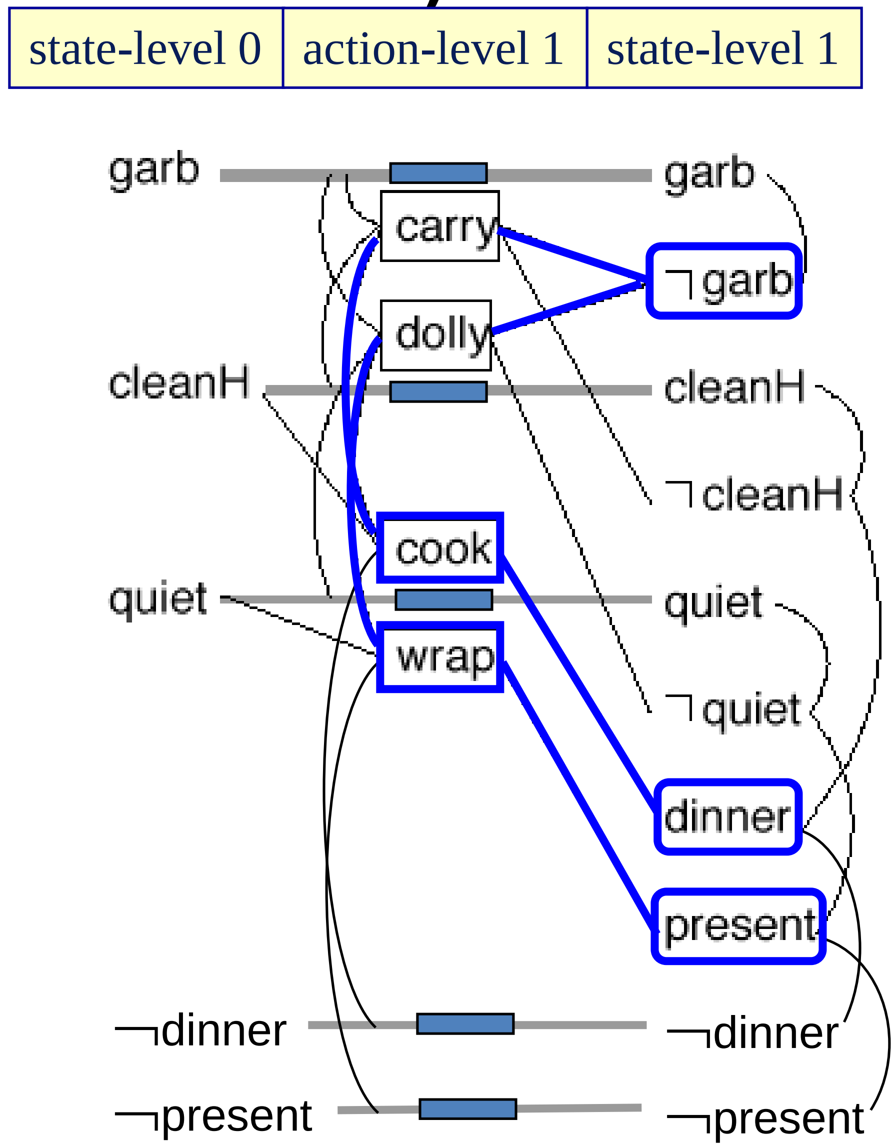
Also have the maintenance actions

# 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



# Example (continued)




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

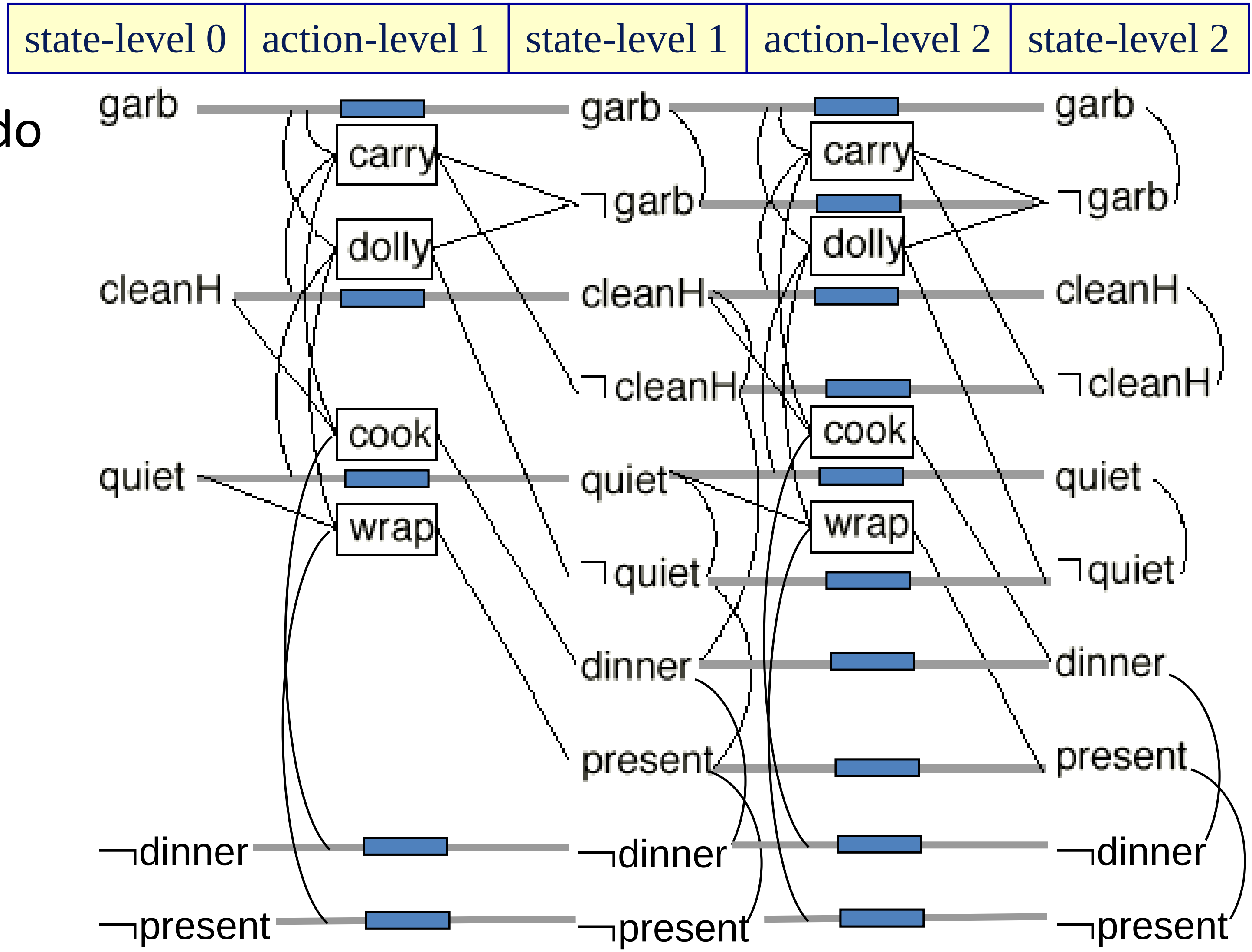
# Recall what the algorithm does

procedure Graphplan:

- for *k* = 0, 1, 2, …
  - *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
- If the graph is stabilized, solution is unreachable

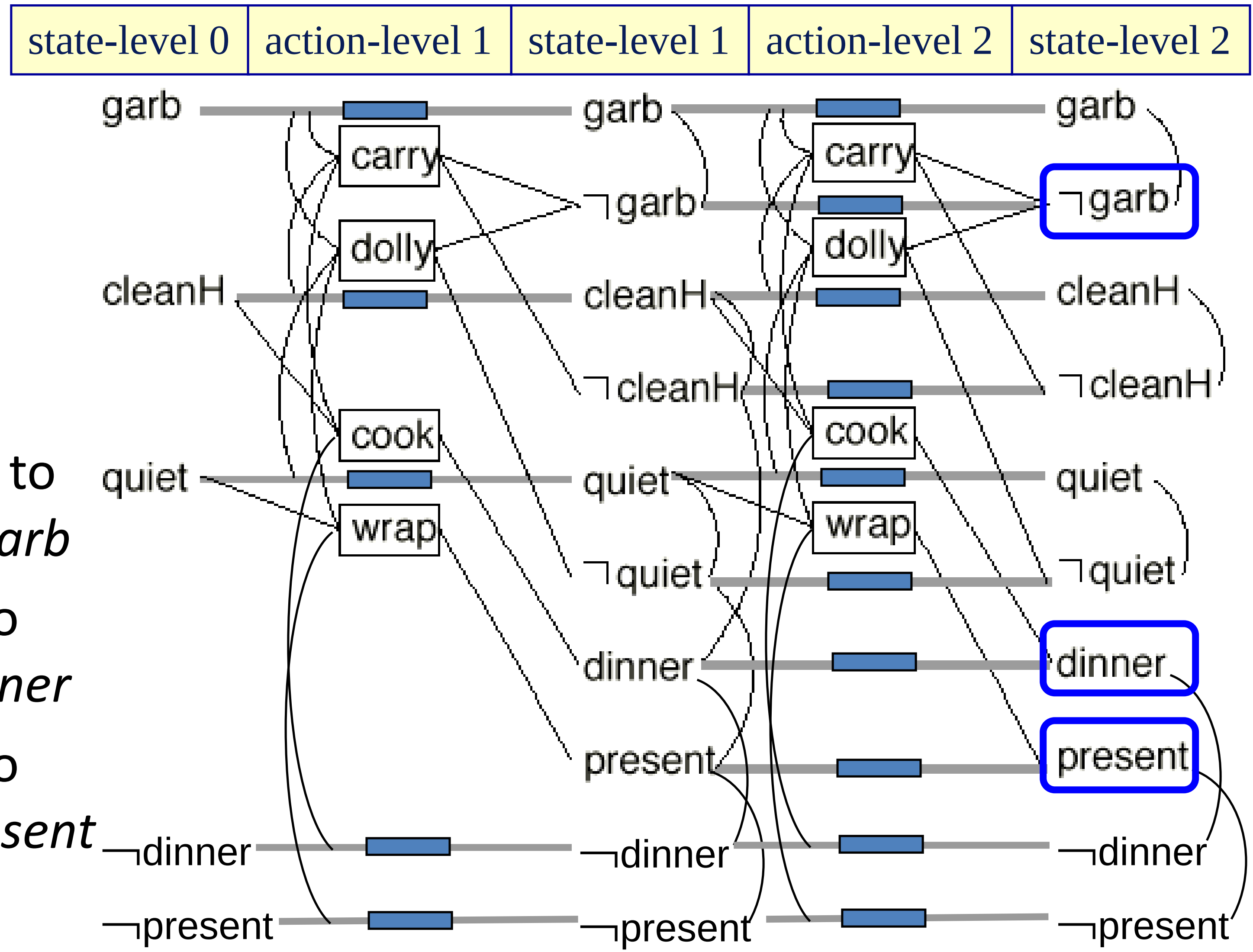
# Example (continued)

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



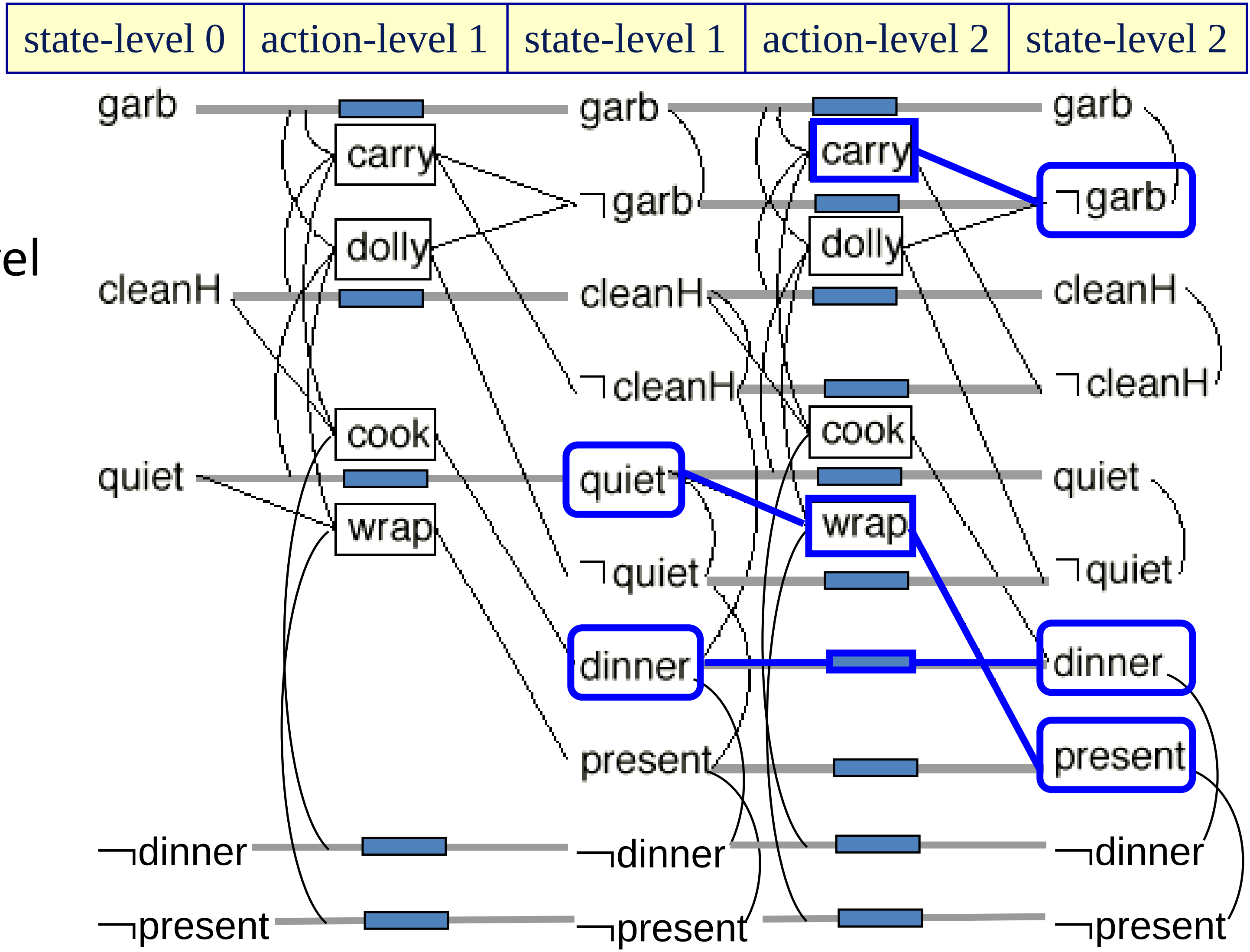
# Example (continued)

- Solution extraction
- Twelve combinations at level 4
  - Three ways to achieve ¬*garb*
  - Two ways to achieve *dinner*
  - Two ways to achieve *present*



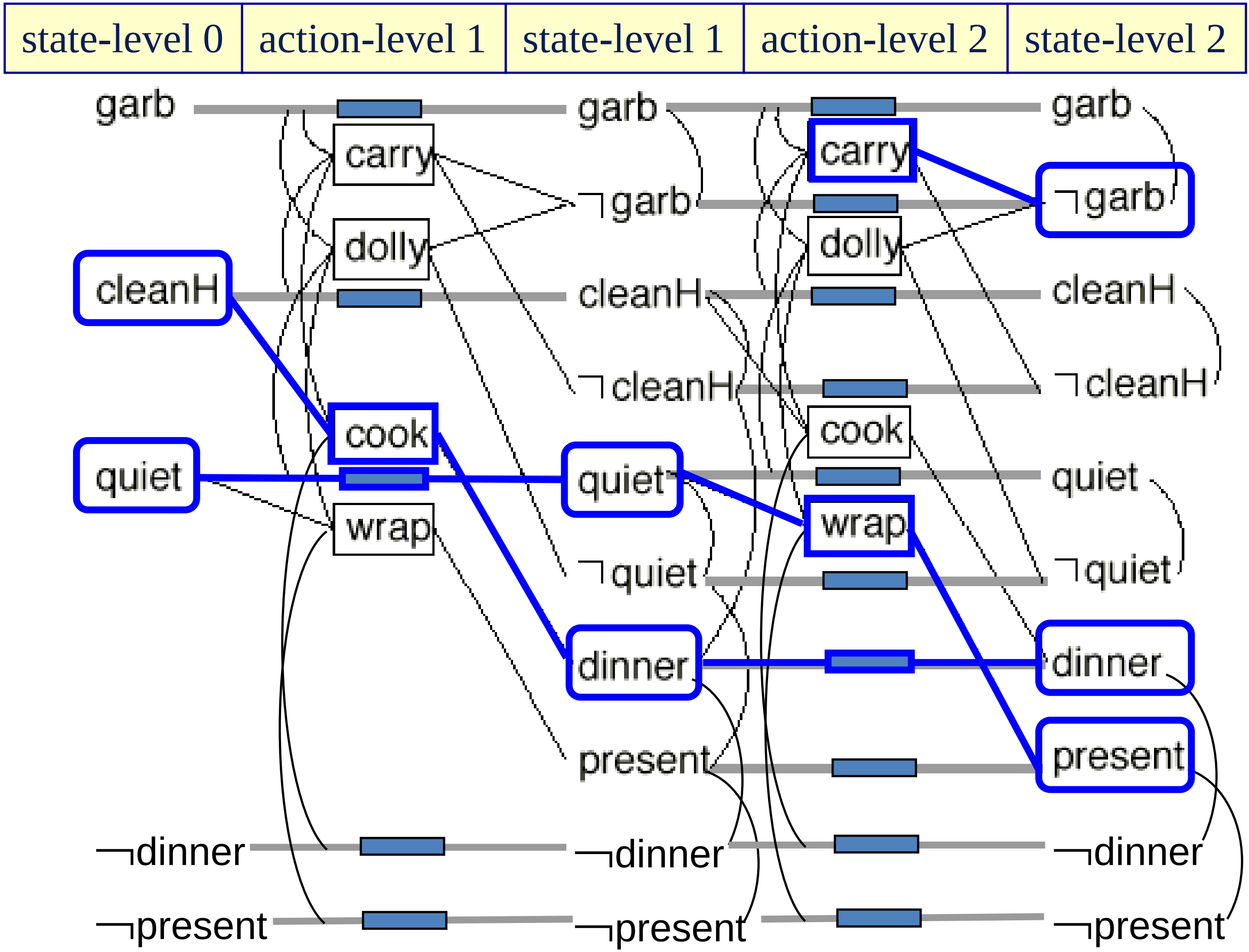
# Example (continued)

- 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