Planning Graphs

– As a search space
– To calculate informed heuristics

Jonas Kvarnström
Automated Planning Group
Department of Computer and Information Science
Linköping University

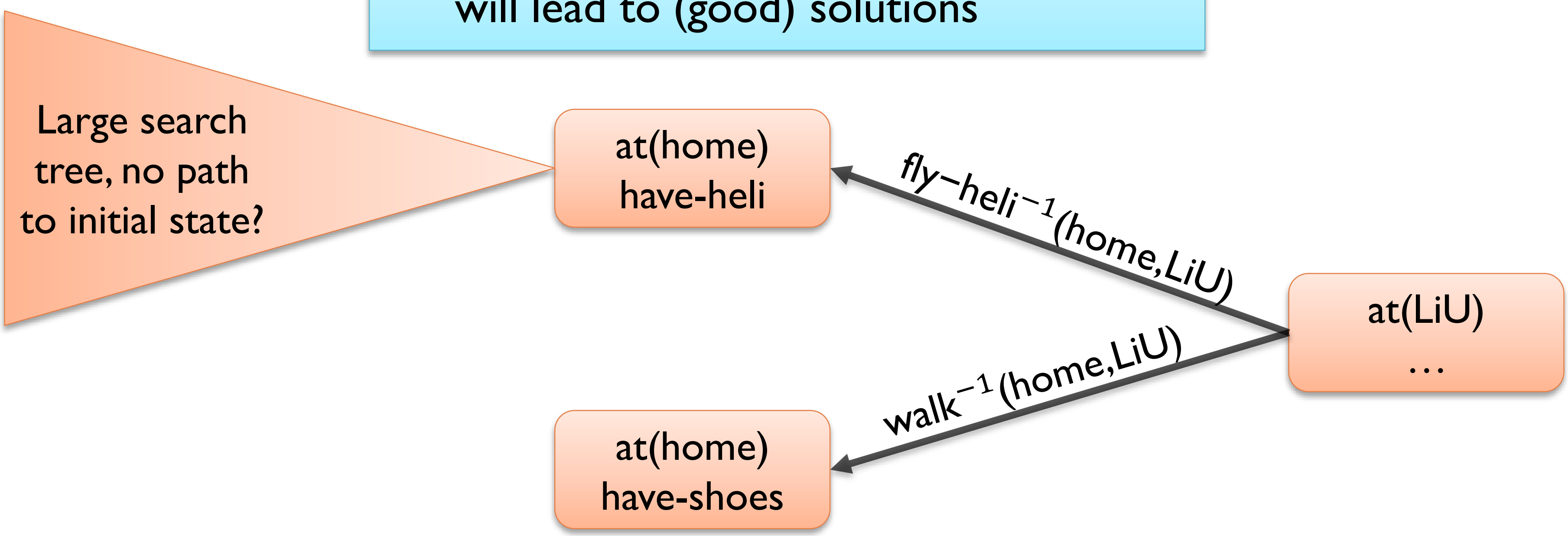
jonas.kvarnstrom@liu.se – 2018

Planning Graphs:
The Underlying Ideas

Recap: Backward Search

BACKWARD SEARCH

- We know if the **effects** of an action can contribute to the goal
- Need **guidance** to determine which backward paths will lead to (good) solutions

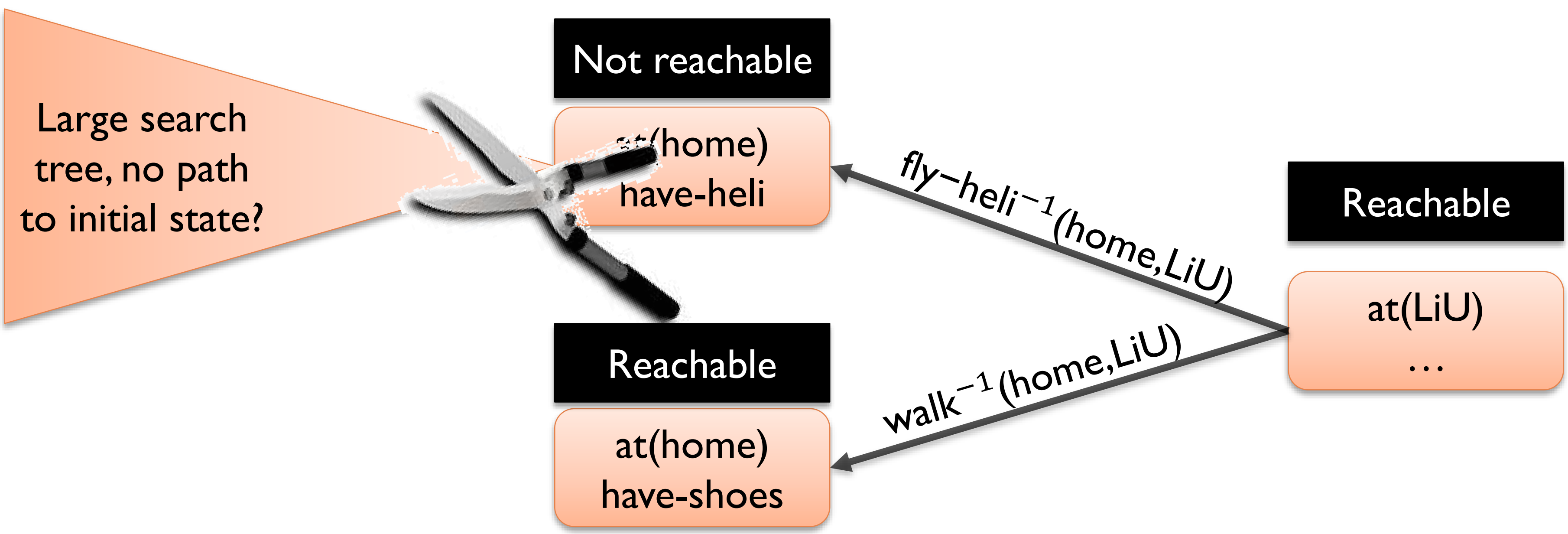


One approach: Use heuristics. But other methods exist…

Reachable States

- Suppose that we could quickly determine:
  - **reachable**(s_0, s) – is state s reachable from s_0?

- Then we could **prune** many "fruitless branches":



But reachable(s_0, s) takes too much time to compute…

Possibly Reachable States

- Instead of exact classification:

Reachable	Not reachable

- Find an approximation: **possibly-reachable**(s_0, s)!

Possibly reachable

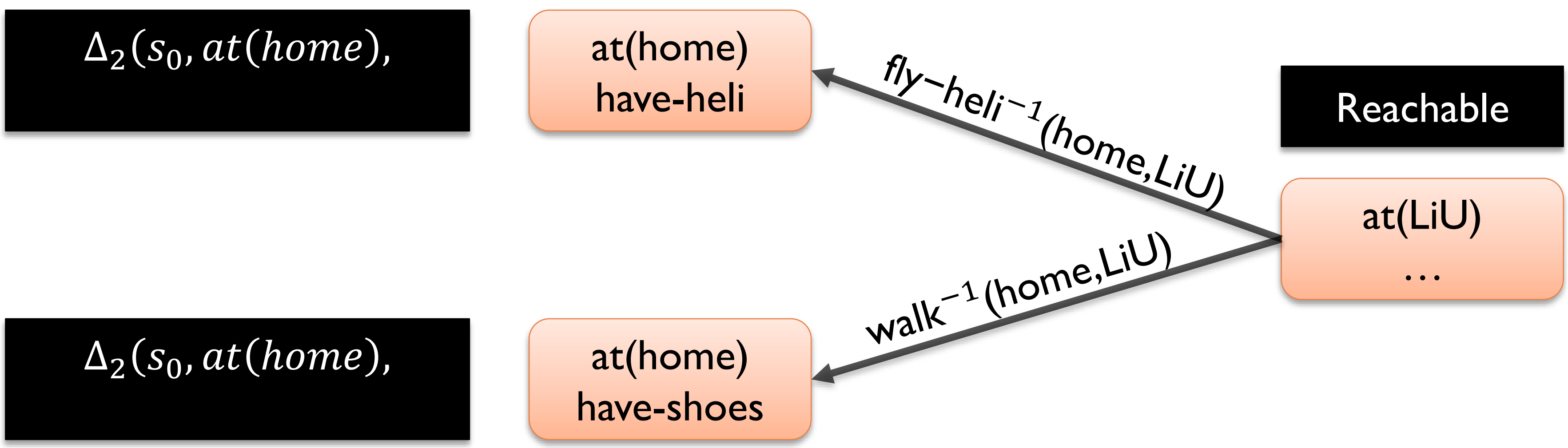
- Actually reachable (not known which ones!)
- Unreachable…

Not (possibly reachable)

- Definitely unreachable…

Possibly Reachable States: Pairwise Mutexes

- Discussing h_2, we saw that if $\Delta_2(s_0, p, q) = \infty$:
  - Starting in s_0, **can't reach any state** where p and q are true
  - Starting in s_0, p and q are *mutually exclusive (mutex)*
  - *Could* use: **possibly-reachable**$(s_0, s) \leftrightarrow \Delta_2(s_0, p, q) \neq \infty$ where p, q in goal



Much better than nothing, but not strong:
(1) only considers pairs p,q that are **never** achievable (2) h_2 does not detect all of those

Possibly Reachable States at Step #i

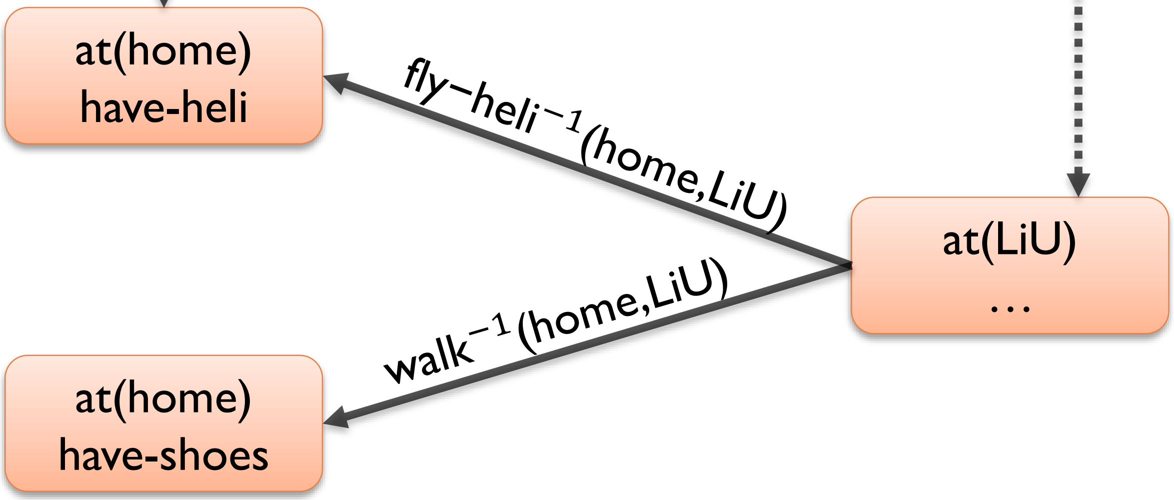
- Improving the accuracy of **possibly-reachable**(s_0, s):
  - Complex – partly because we must consider paths of ***arbitrary*** length
  - Instead: Apply ideas from ***iterative deepening search***
    - Is there a plan of length 0? Of length 1? Length 2? 3? 4? …

Possibly Reachable States: Example

- **possibly-reachable-at-step**$(s_0, 0, s)$? No!
- **possibly-reachable-at-step**$(s_0, 1, s)$? No!
- **possibly-reachable-at-step**$(s_0, 2, s)$? No!
- **possibly-reachable-at-step**$(s_0, 3, s)$? No!
- **possibly-reachable-at-step**$(s_0, 4, s)$? No!

possibly-reachable-at-step$(s_0, 5, s)$? Yes!

possibly-reachable-at-step$(s_0, 4, s)$?
Hopefully no (prune), maybe yes (search)!

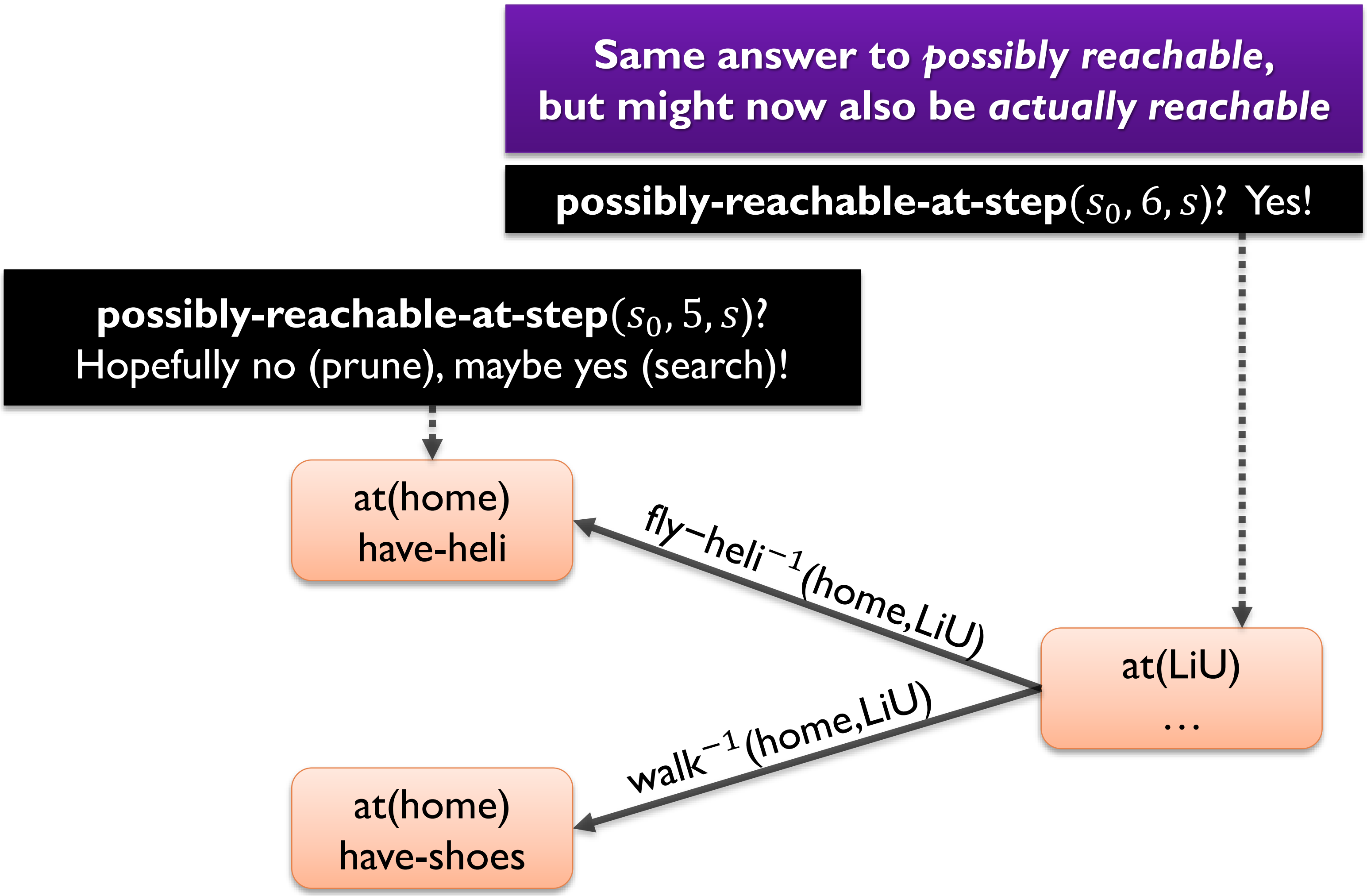


Might not *actually* find a solution in 5 steps (**possibly** reachable!)

Prunes some parts, but search still needed

Keep Iterating

- If no solution was found:
  - Keep going with plan length 6, 7, …

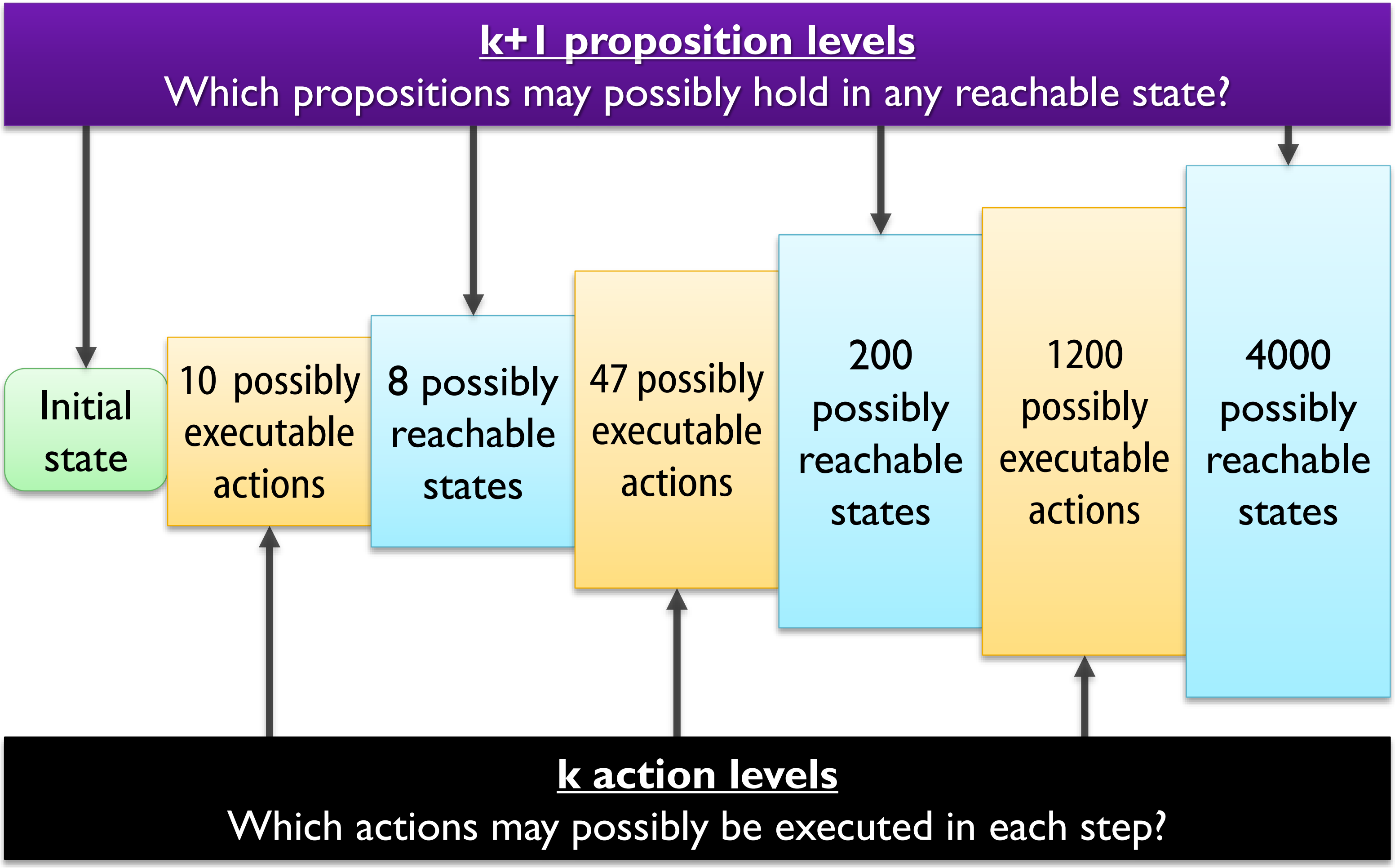


Planning Graphs and the GraphPlan Planner

An efficient representation
for *possibly reachable states*

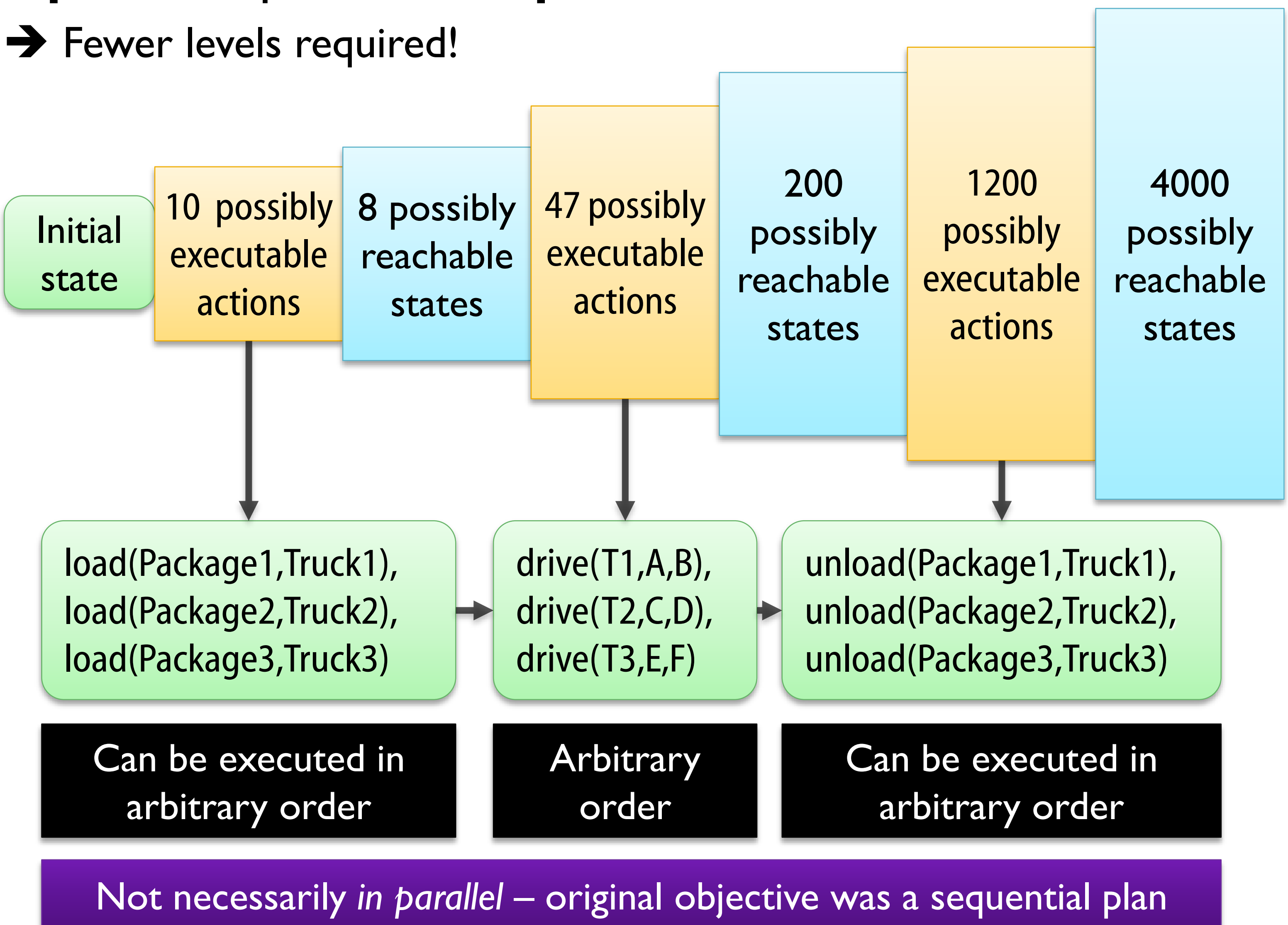
Planning Graph

- A **Planning Graph** also considers **possibly executable actions**
  - Useful to *generate states* – also useful in *backwards search!*



GraphPlan: Plan Structure

- **<u>GraphPlan</u>**'s plans are **<u>sequences</u>** of **<u>sets</u>** of actions
  - ➔ Fewer levels required!



Running Example

- **Running example** due to Dan Weld (modified):
  - Prepare and serve a surprise dinner,
    take out the garbage,
    and make sure the present is wrapped before waking your sweetheart!

s_0 = {clean, garbage, asleep}

g = {clean, ¬garbage, served, wrapped}

Action	Preconds	Effects
cook()	clean	dinner
serve()	dinner	served
wrap()	asleep	wrapped
carry()	garbage	¬garbage, ¬clean
roll()	garbage	¬garbage, ¬asleep
clean()	¬clean	clean

Reachable States

- Suppose we actually computed all reachable states
  - Time 0:
    - s_0 ➔ {clean, garbage, asleep}
  - Time 1:
    - cook ➔ {clean, garbage, asleep, **dinner**}
    - serve ➔ *impossible*
    - wrap ➔ {clean, garbage, asleep, **wrapped**}
    - carry ➔ {asleep}
    - roll ➔ {clean}
    - clean ➔ *impossible*
    - cook+wrap ➔ {garbage, clean, asleep, **dinner, wrapped**}
    - cook+roll ➔ {clean, **dinner**}
    - …
  - Time 2:
    - cook/cook ➔ {clean, garbage, asleep, dinner}
    - cook/serve ➔ {clean, garbage, asleep, dinner, served}
    - cook/wrap ➔ {clean, garbage, asleep, dinner, wrapped}
    - cook/carry ➔ {asleep, dinner}
    - cook/roll ➔ {clean, dinner}
    - cook/clean ➔ not possible
    - wrap/cook ➔ …

Let's calculate **reachable literals** instead!

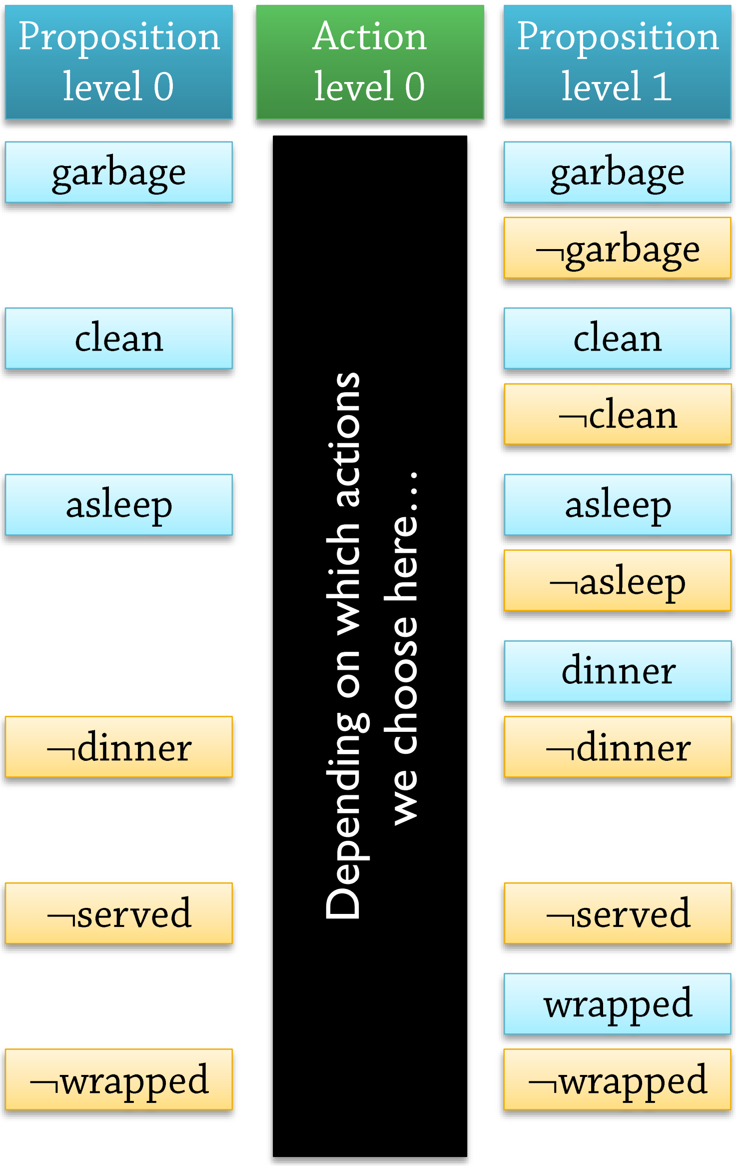
Reachable Literals (1)

- Time 0:
  - ➔ s_0 = {garbage, clean, asleep}
- Time 1:
  - cook ➔ s_1 = {garbage, clean, asleep, dinner}
  - *(serve)* *not applicable*
  - wrap ➔ s_2 = {garbage, clean, asleep, wrapped}
  - carry ➔ s_3 = {asleep}
  - roll ➔ s_4 = {clean}
  - *(clean)* *not applicable*

No need to consider **sets** of actions:
All literals made true
by any combination of "parallel" actions
are already there

We can't reach all **combinations**
of literals in proposition level 1…

But we can reach **only** such combinations!
Can *not* reach a state where served is true

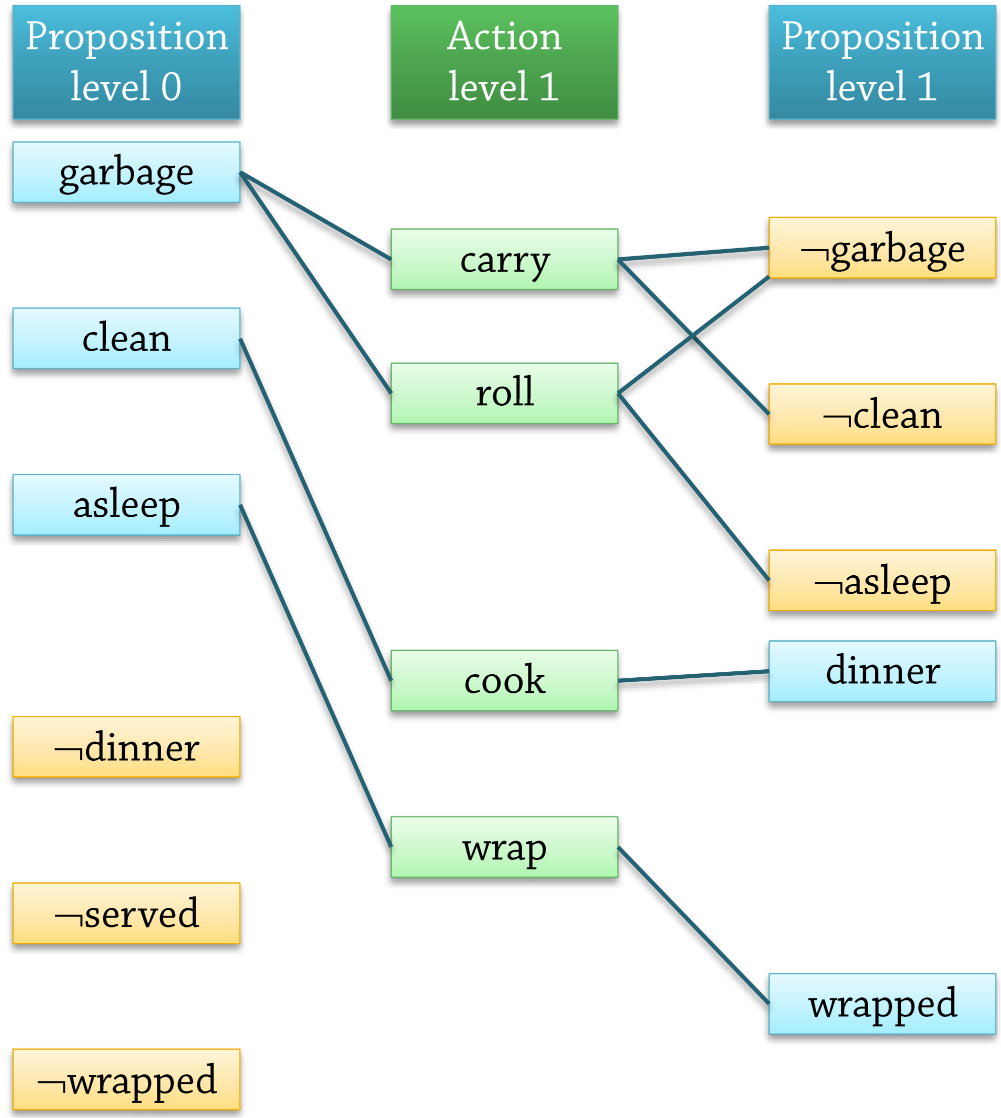


Reachable Literals (2)

- Planning Graph Extension:
  - Start with one "prop level"
    - Set of reachable literals
  - For each **applicable action**
    - Add its **effects** to the next proposition level
    - Add **edges** to preconditions and to effects for **bookkeeping** (used later!)

Action	**Precond**	**Effects**
cook()	clean	dinner
serve()	dinner	served
wrap()	asleep	wrapped
carry()	garbage	¬garbage, ¬clean
roll()	garbage	¬garbage, ¬asleep
clean()	¬clean	clean

But wait!
Some propositions are missing…

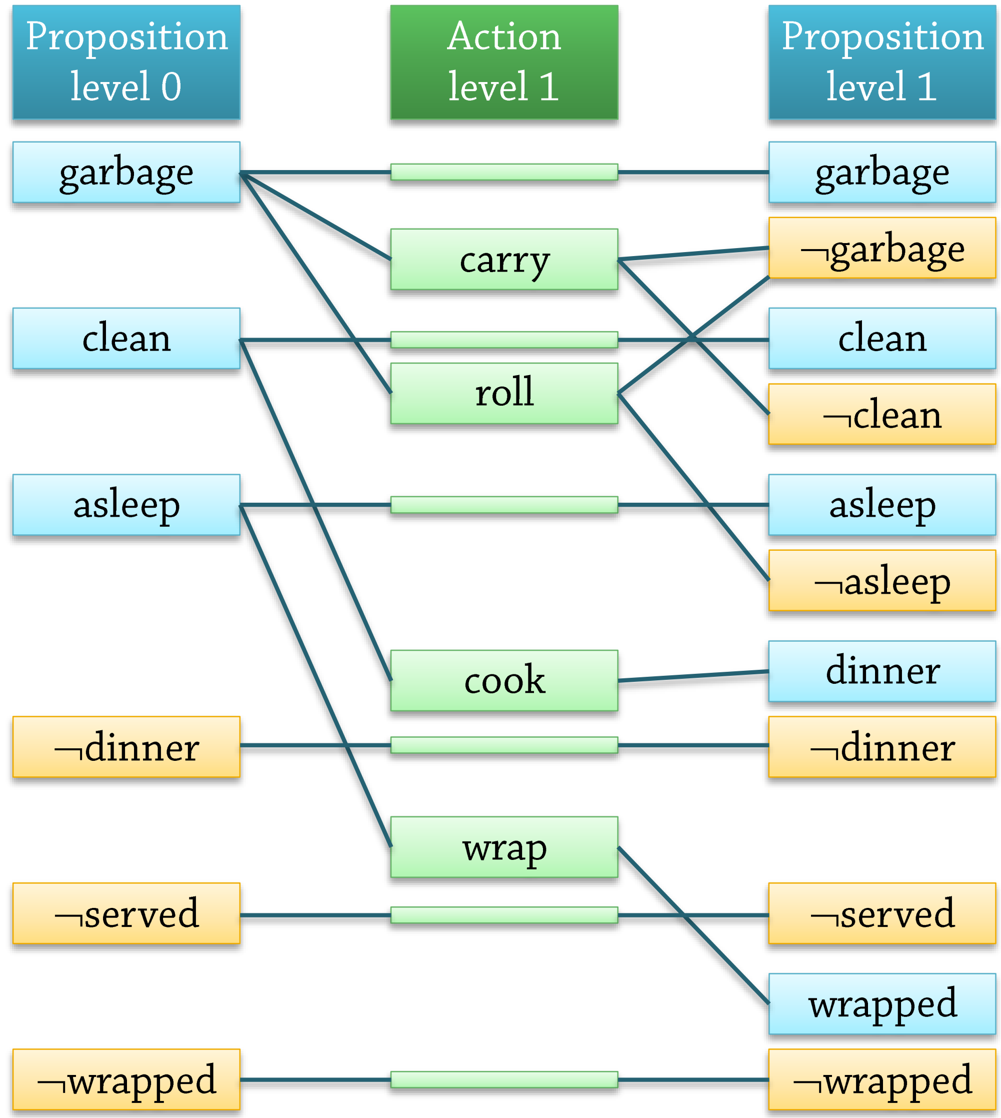


Reachable Literals (3)

- Depending on the actions chosen, facts could **persist** from the previous level!

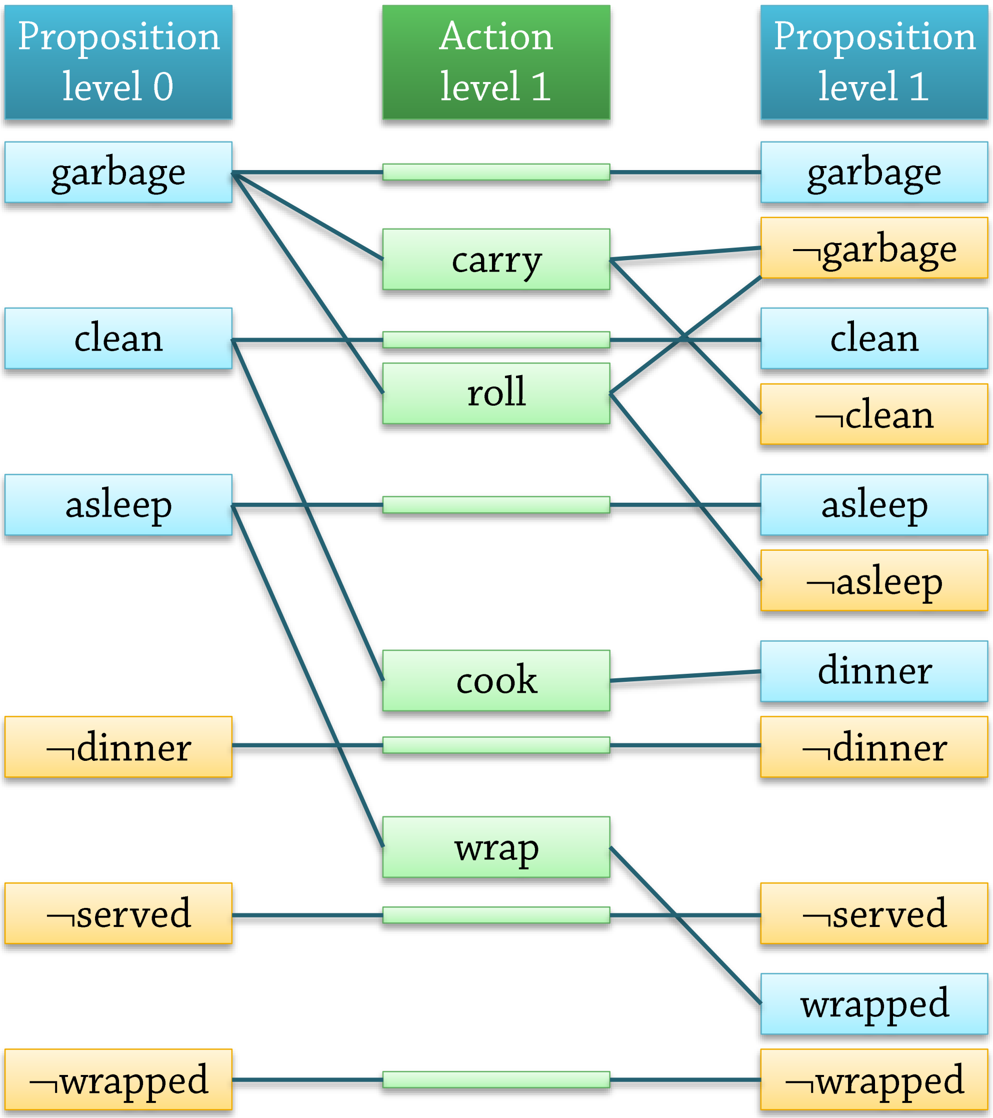
- To handle this **consistently**: **maintenance (noop) actions**
  - One for each literal *l*
  - Precond = effect = *l*

Action	**Precond**	**Effects**
cook()	clean	dinner
serve()	dinner	served
wrap()	asleep	wrapped
carry()	garbage	¬garbage, ¬clean
roll()	garbage	¬garbage, ¬asleep
clean()	¬clean	clean
noop-dinner	dinner	dinner
noop-¬dinner	¬dinner	¬dinner
…		



Reachable Literals (4)

- Now the graph is **sound**
  - If an action **might** be executable in step *n*, it is part of the graph
  - If a literal **might** hold after *n* actions, it is part of the graph
- But it is quite "**weak**"!
  - Even at proposition level 1, it seems **any** literal except ***served*** can be achieved
- We need more information
  - In an ***efficiently useful*** format
  - **Mutual exclusion**



Mutual Exclusion

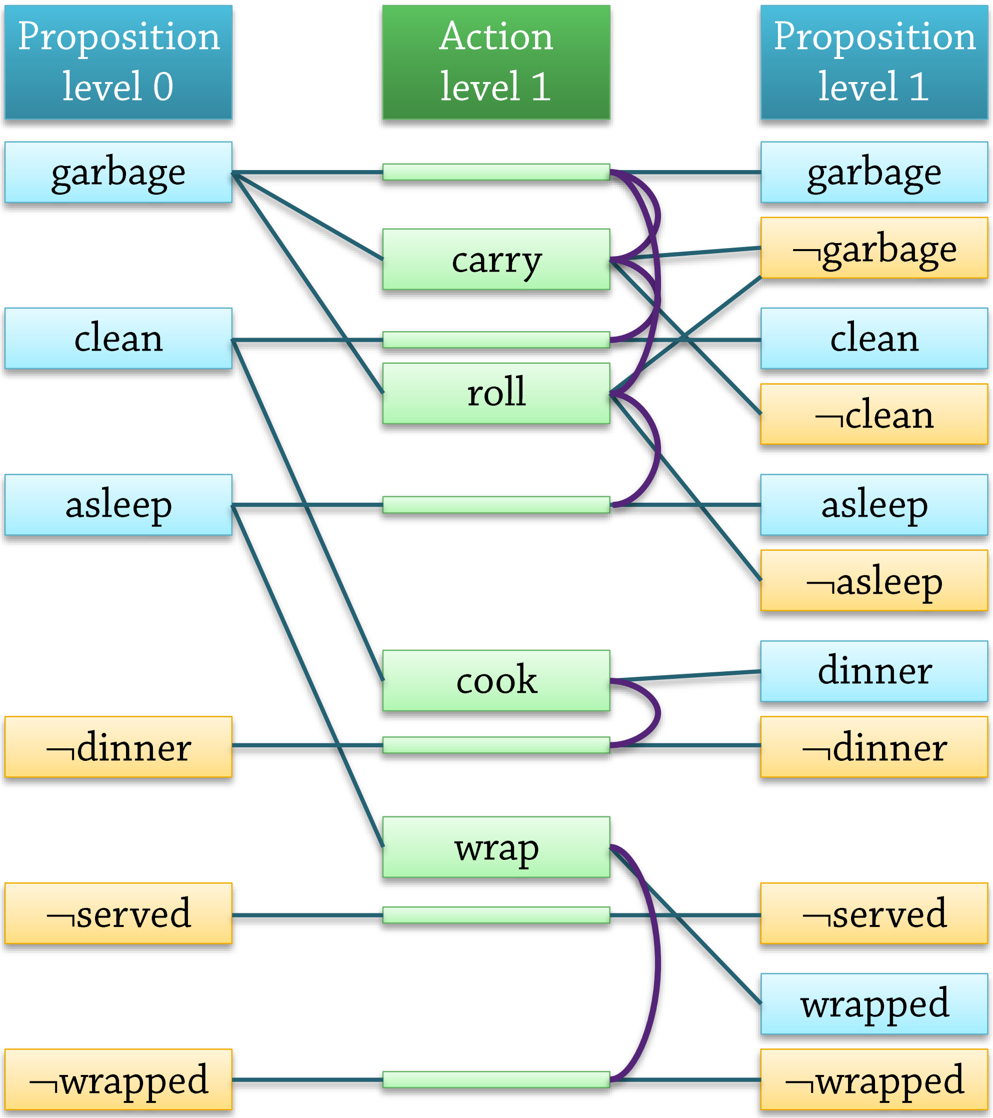
Mutex 1: Inconsistent Effects

No mutexes at proposition level 0:
We assume a *consistent* initial state!

Two **actions** in a level are mutex
if their **effects are inconsistent**

Can't execute them in parallel, and
order of execution is not arbitrary

- *carry* / noop-*garbage*,
  carry / noop-*clean*
  - One causes garbage,
    the others cause *not* garbage
- *roll* / noop-*garbage*,
  roll / noop-*asleep*
- *cook* / noop-¬*dinner*
- *wrap* / noop-¬*wrapped*

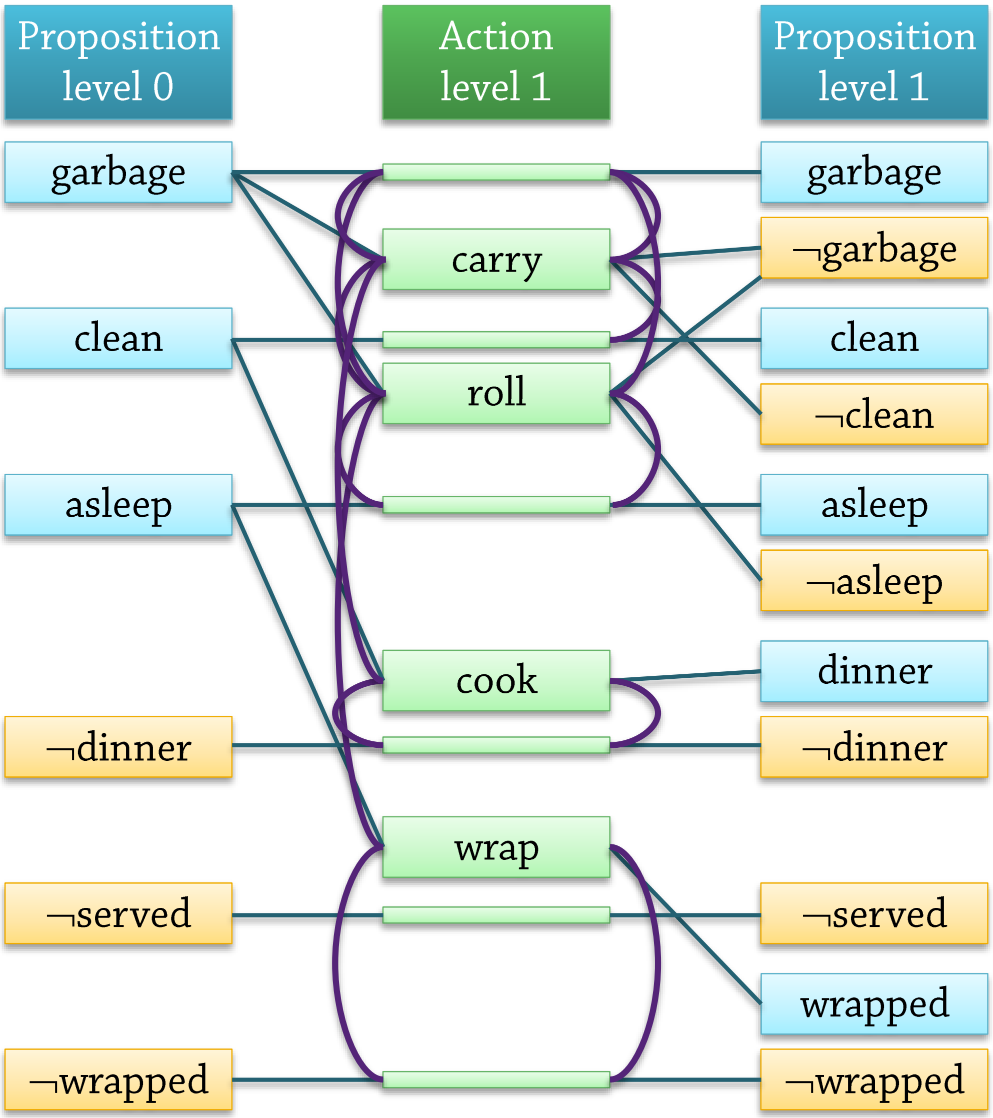


Mutex 2: Interference

Two **actions** in one level are mutex if **one destroys a precondition** of the other

Can't be executed in arbitrary order

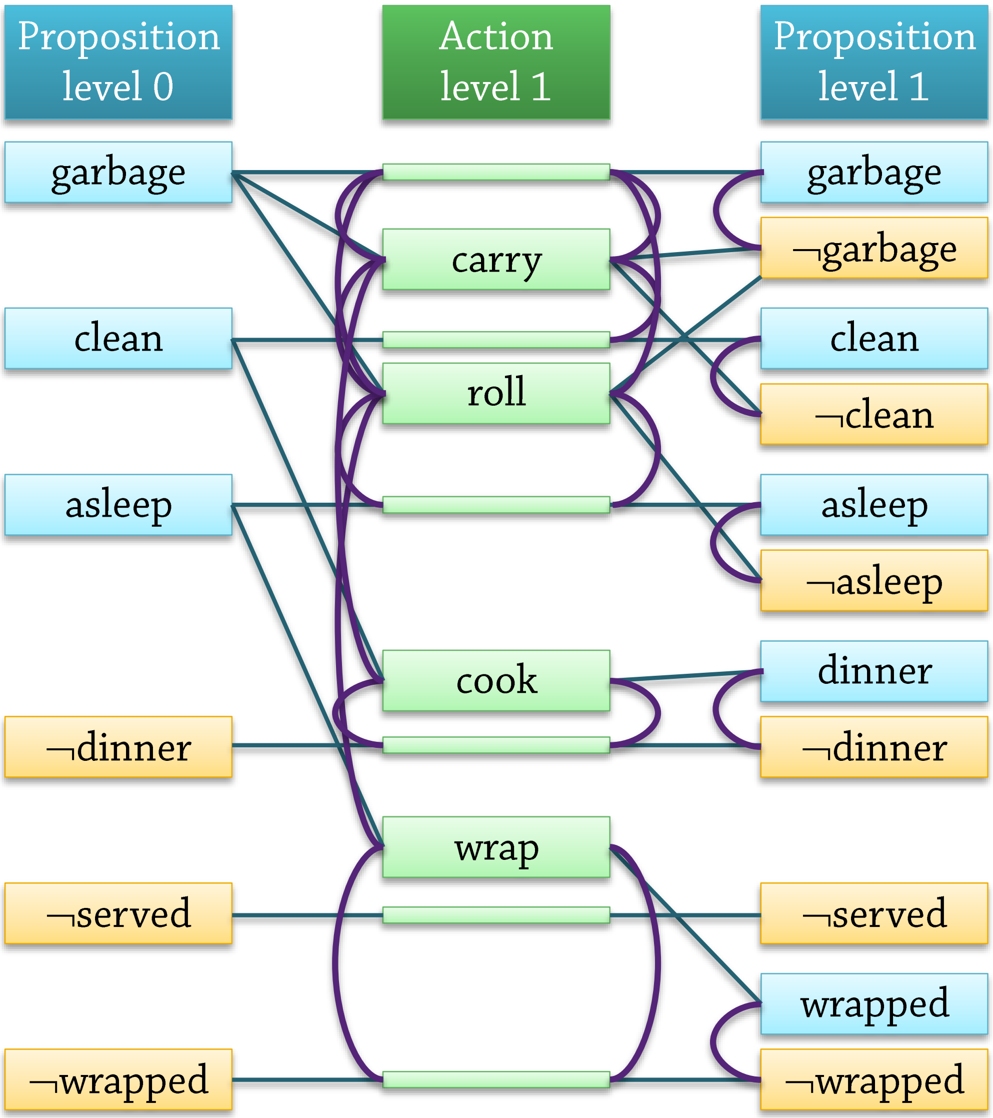
- *roll* is mutex with *wrap*
  - *roll* deletes *asleep*
  - *wrap* needs *asleep*
- *carry* is mutex with *noop-garbage*
  - *carry* deletes *garbage*
  - *noop-garbage* needs *garbage*
- …



Mutex 3: Inconsistent Support (A)

Two **propositions** are mutex
if one is the **negation** of the other

Can't be true at the same time…



Mutex 4: Inconsistent Support (B)

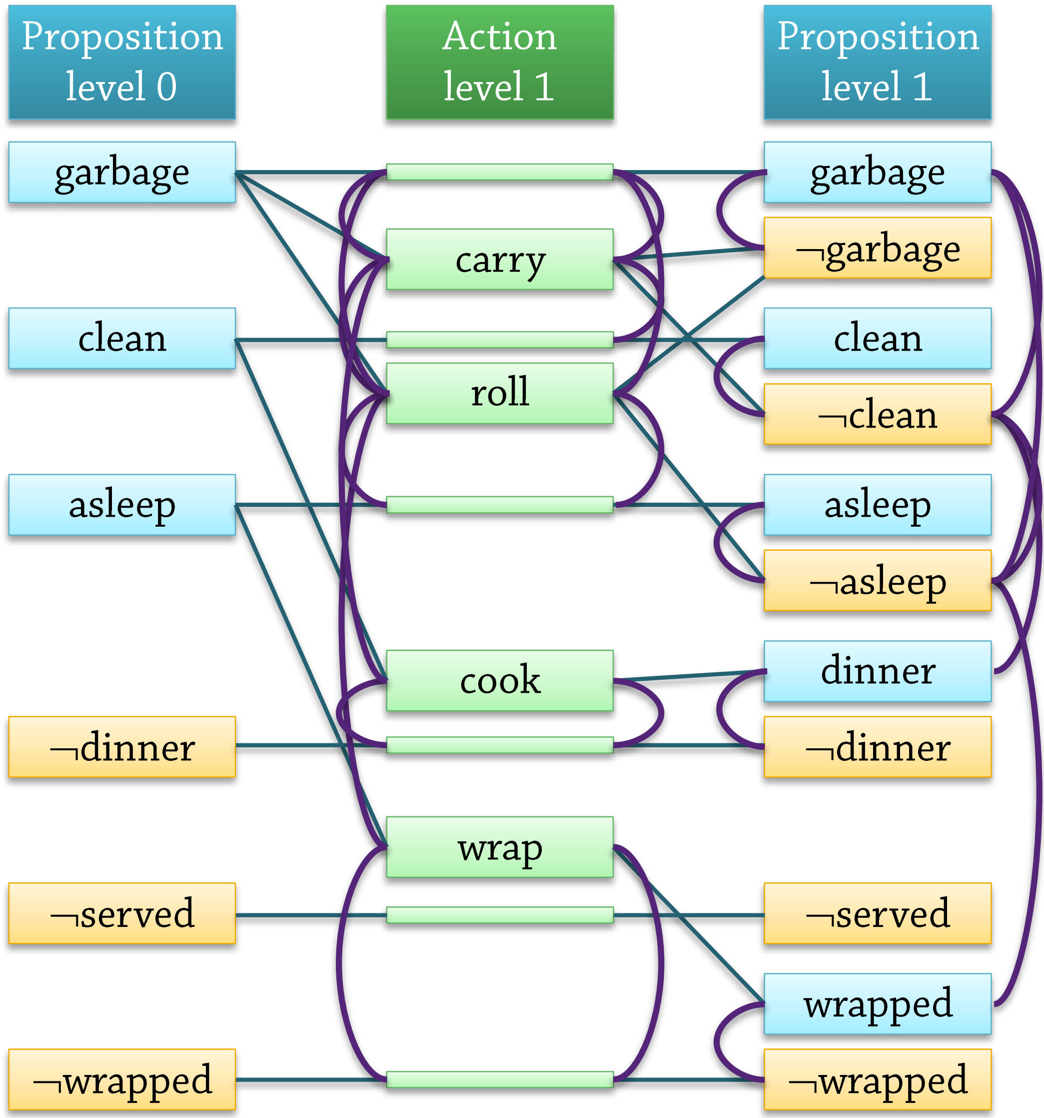
Two **propositions** are mutex
if they have **inconsistent support**

All actions that achieve them
are pairwise mutex
in the previous level

¬asleep can only be achieved by *roll*,
wrapped can only be achieved by *wrap*,
and *roll*/*wrap* are mutex
➔ *¬asleep* and *wrapped* are mutex

¬clean can only be achieved by *carry*,
dinner can only be achieved by *cook*,
and *carry*/*cook* are mutex
➔ *¬clean* and *dinner* are mutex

¬garbage can be achieved by *roll*,
clean can be achieved by *noop-clean*,
and *roll*/*noop-clean* are **not** mutex
➔ *¬garbage* and *clean* are **not** mutex

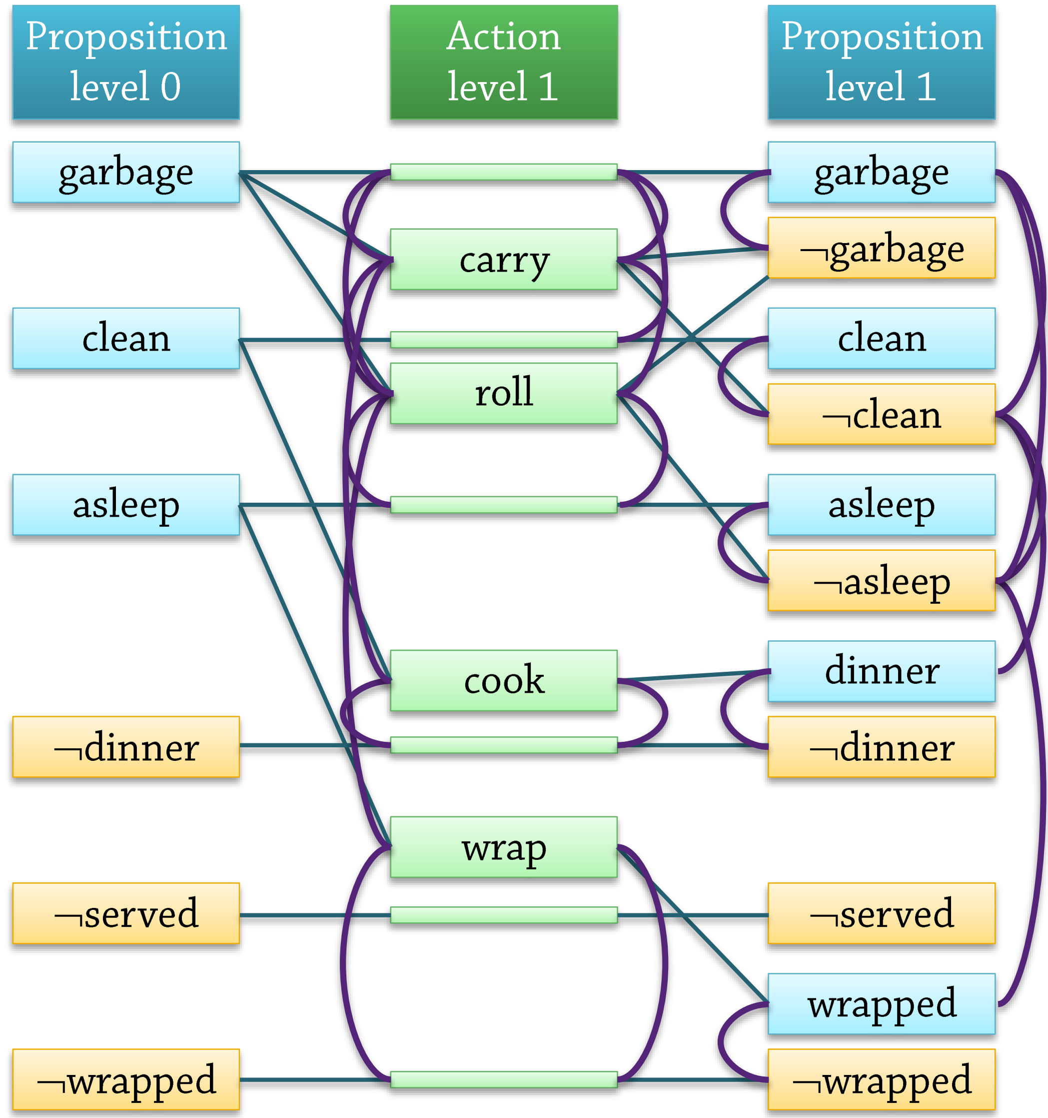


Mutexes: Only pairwise

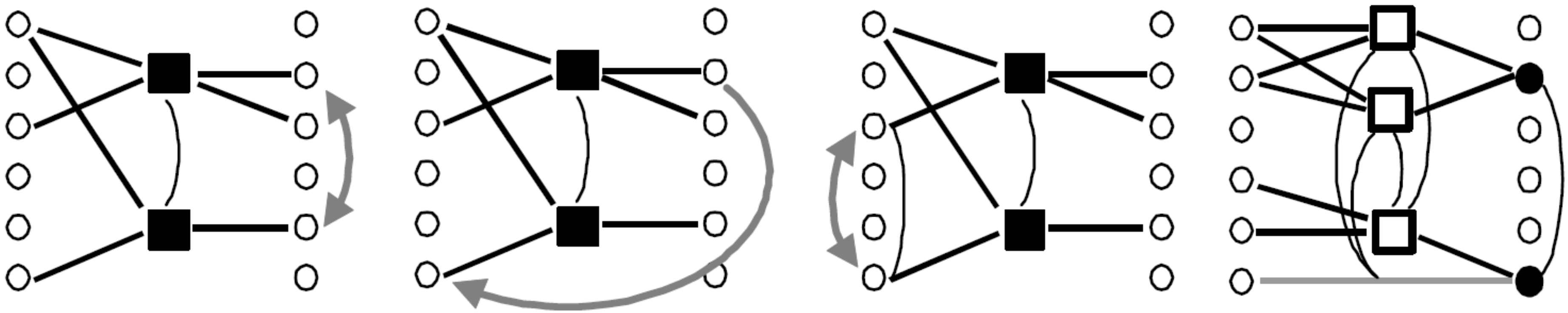
Note:

In reality you cannot have
{ ¬*garbage*, *clean*, *quiet* }
after a single (non-mutex)
action level!

<u>Not detected</u>:
Examining *triples* is more expensive
and not worth the cost



Mutual Exclusion: Overview



Inconsistent Effects | Interference | Competing Needs | Inconsistent Support

- Two **<u>actions</u>** 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 (*not shown*)
- Otherwise:
  - Both might appear at the same time step in a solution plan

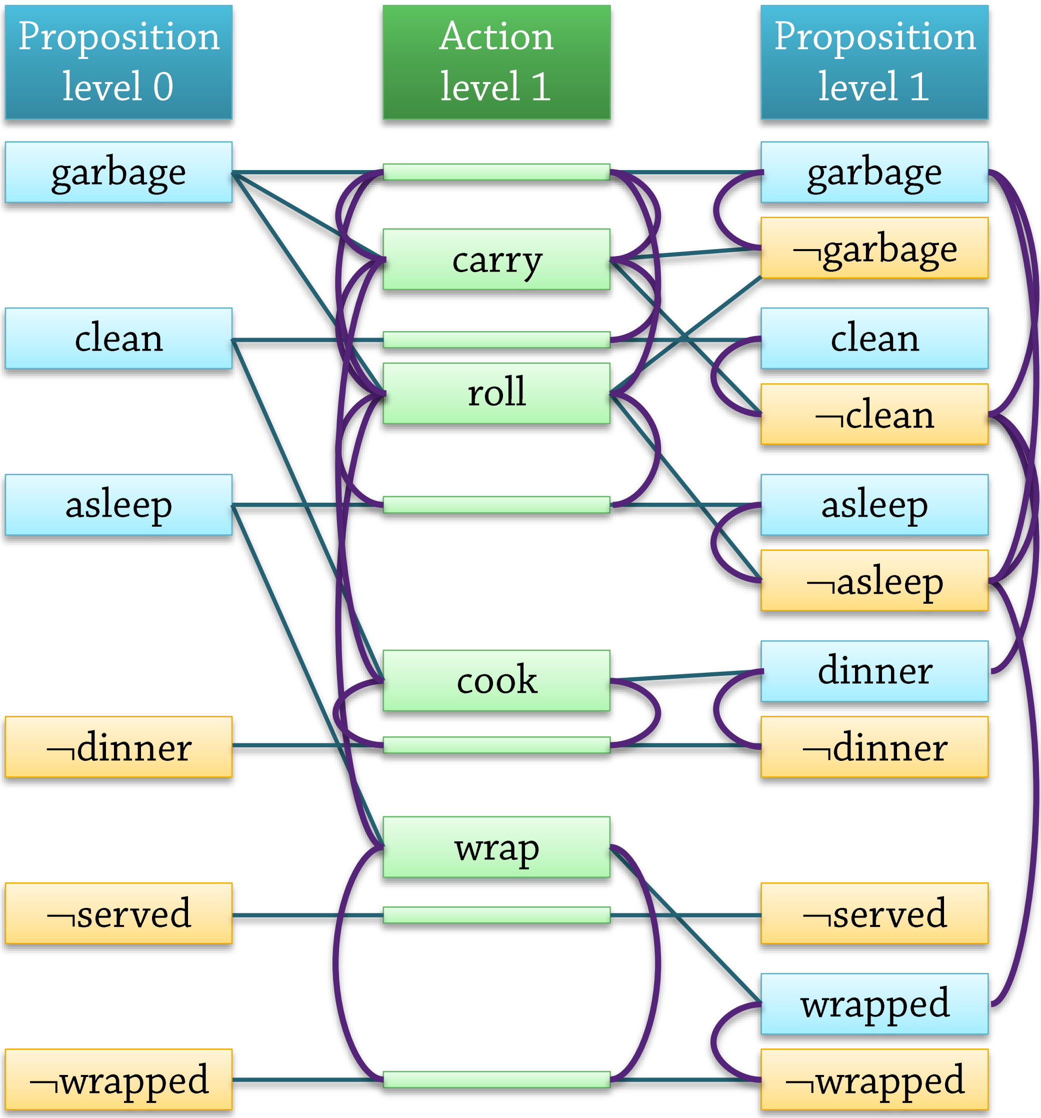
- Two **<u>literals</u>** at the same proposition level are mutex if
  - *Inconsistent support A:* one is the negation of the other,
  - *Inconsistent support B:* **all ways of achieving them are pairwise mutex**

Recursive propagation of mutexes

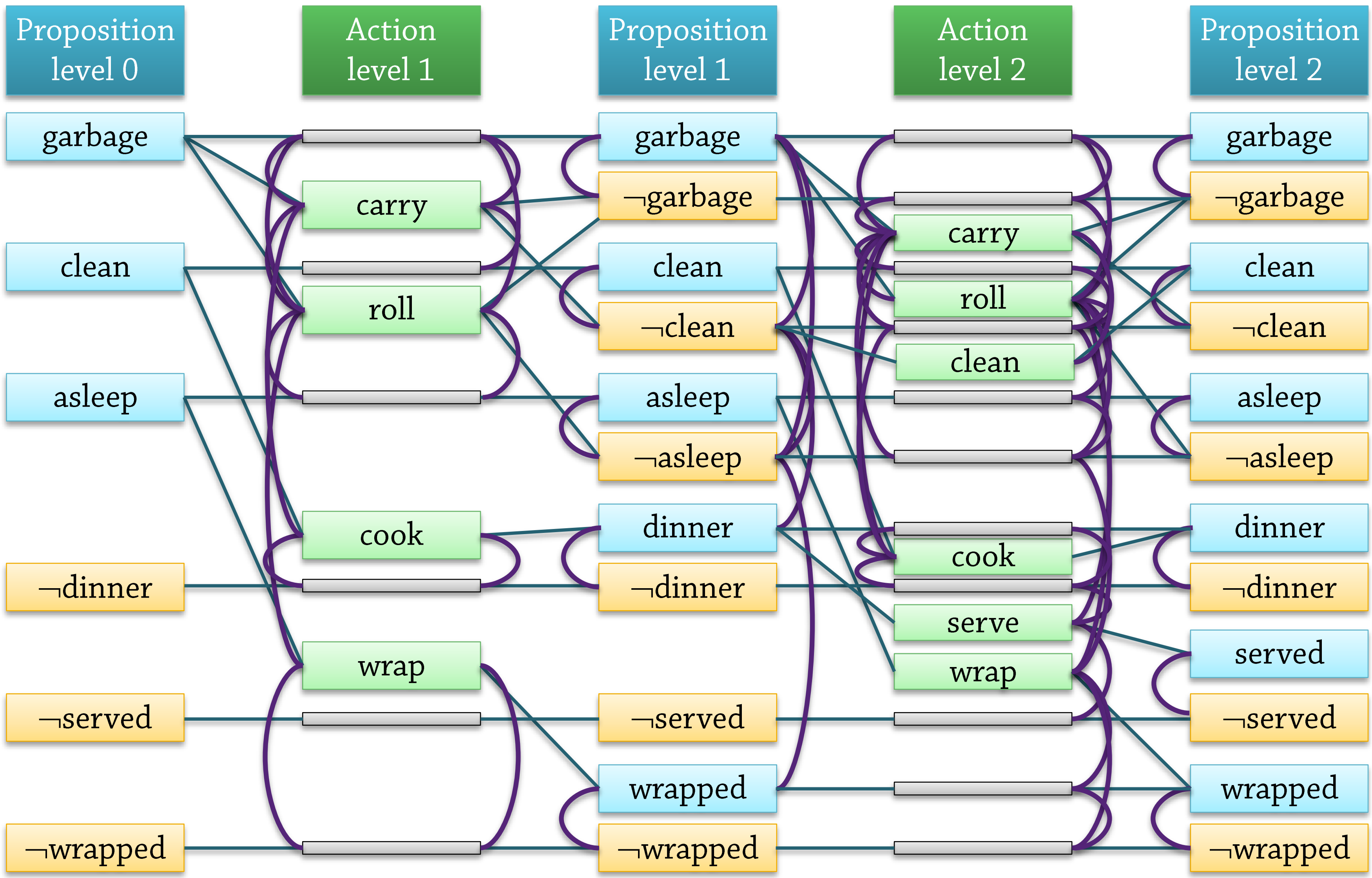
Solution Test

Early Solution Check

- Is there a possible solution?
  - **Goal** g = {clean, ¬garbage, served, wrapped}
  - No: Cannot reach a state where served is true in a single (multi-action) step



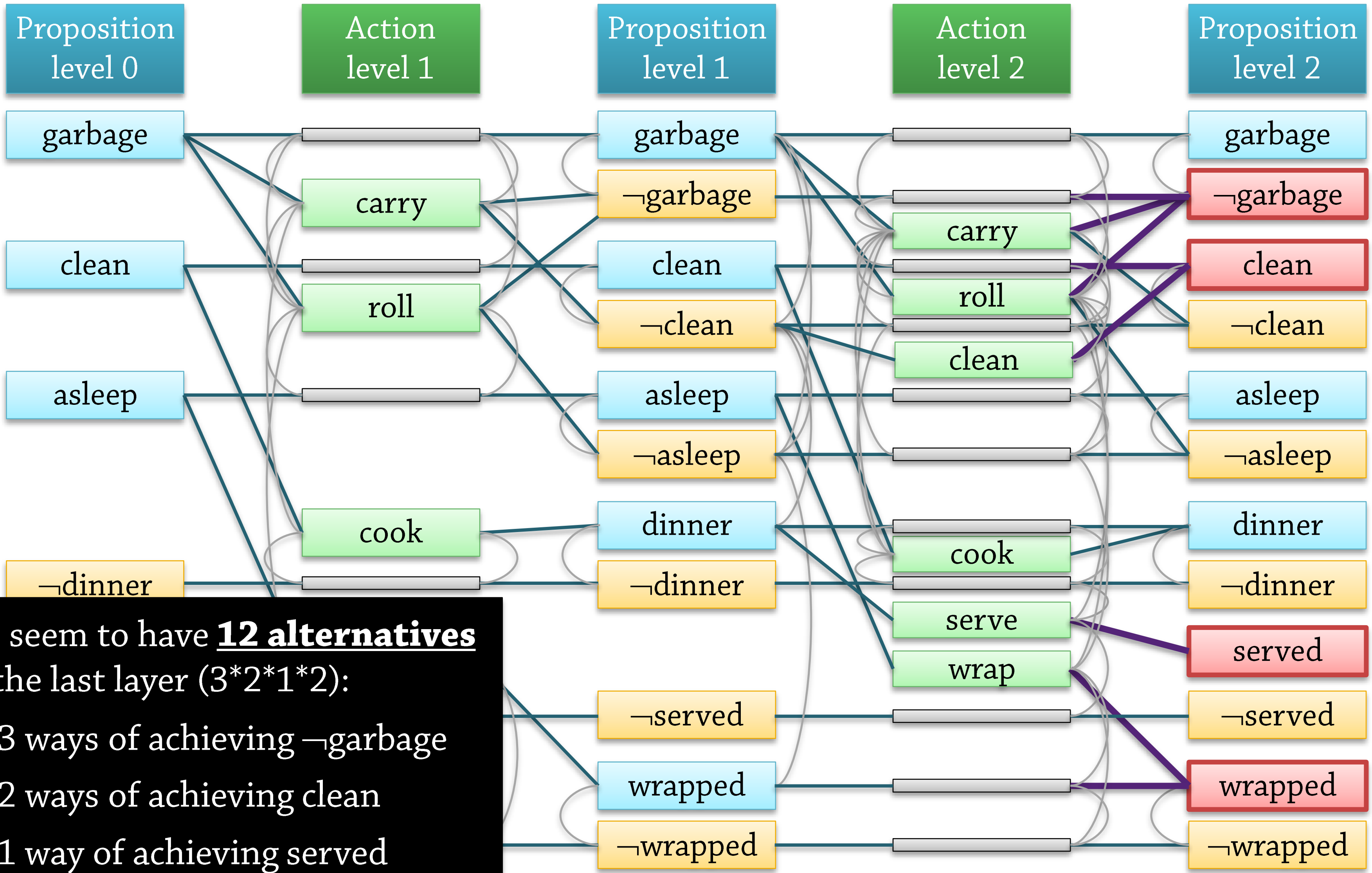
Expanded Planning Graph



All goal literals are present in level 2, and none of them are (known to be) mutex!

Solution Extraction: Backward Search

Solution Extraction (1)

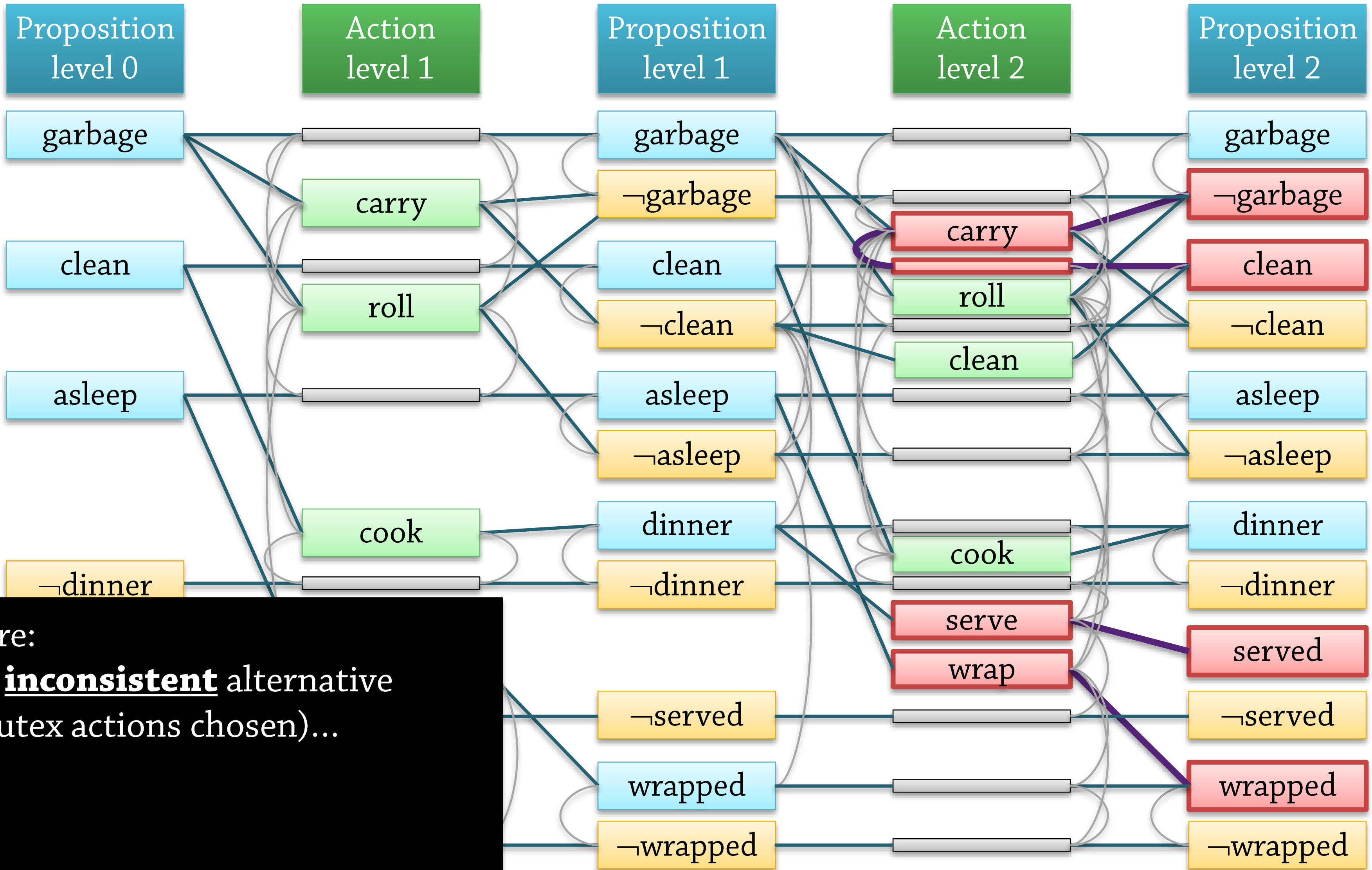


We seem to have **12 alternatives** at the last layer (3*2*1*2):

- 3 ways of achieving ¬garbage
- 2 ways of achieving clean
- 1 way of achieving served
- 2 ways of achieving wrapped

g = {clean, ¬garbage, served, wrapped}

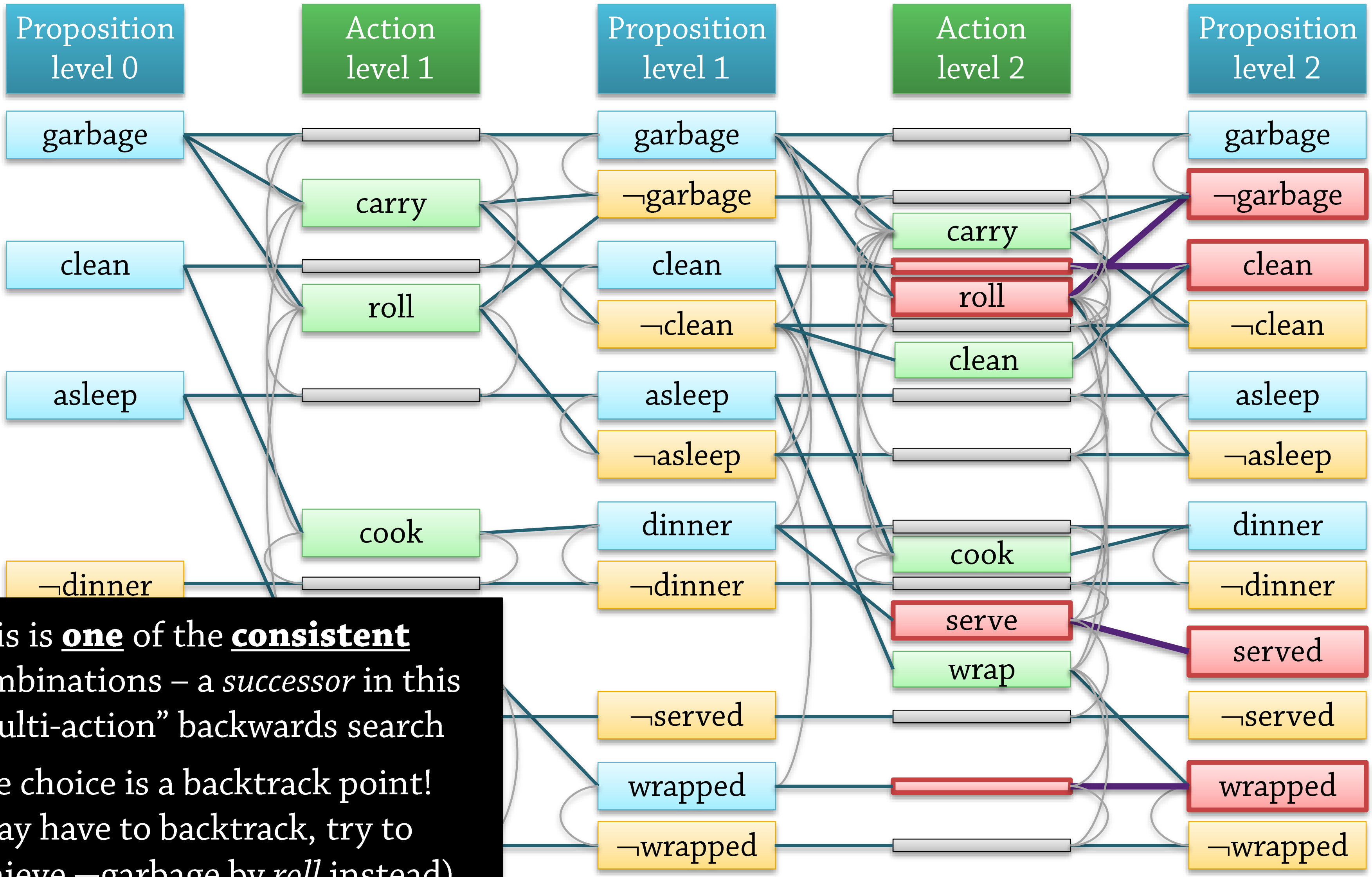
Solution Extraction (2)



Here:
An **inconsistent** alternative
(mutex actions chosen)…

g = {clean, ¬garbage, served, wrapped}

Solution Extraction (3)

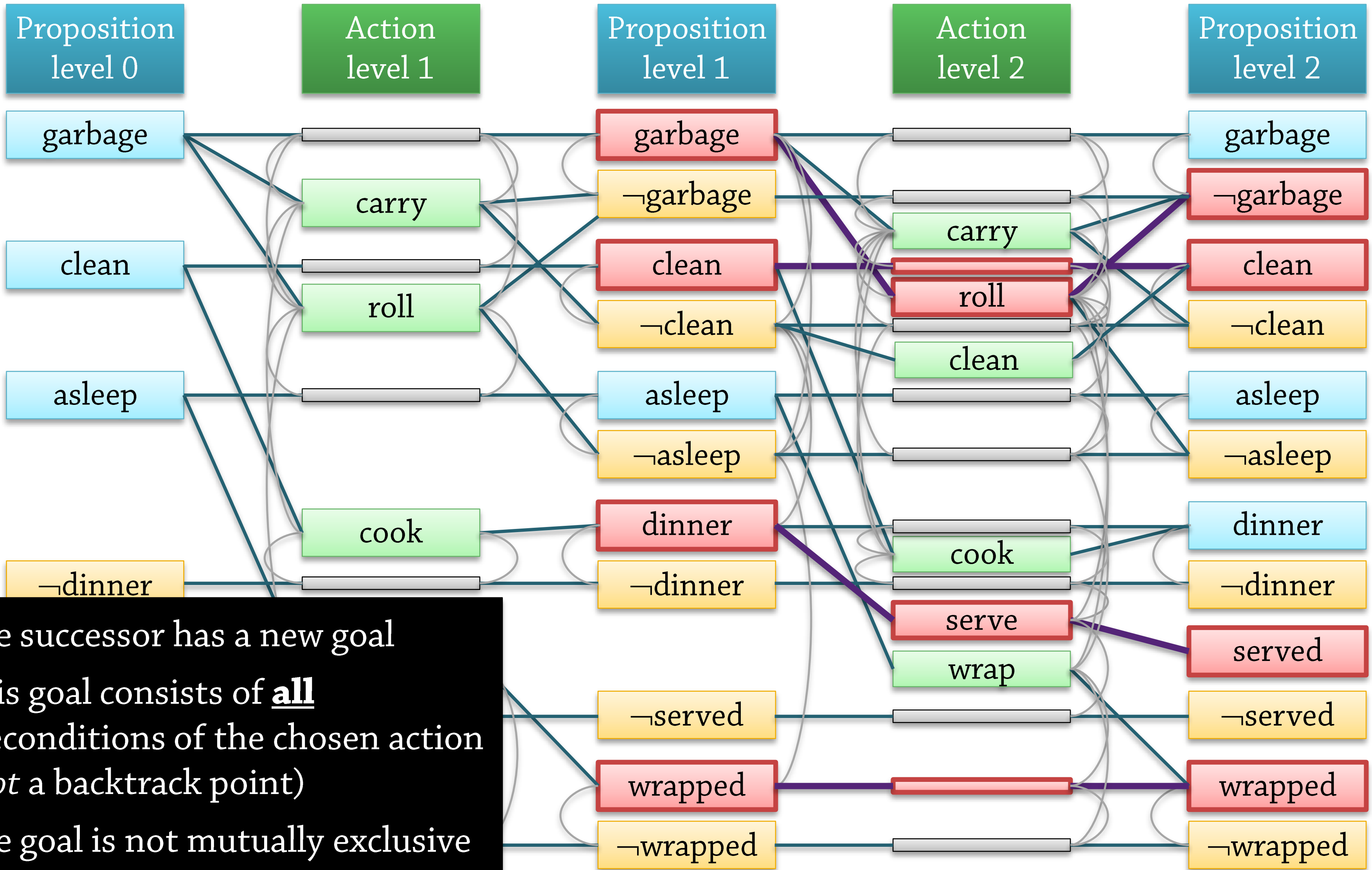


This is **one** of the **consistent** combinations – a *successor* in this "multi-action" backwards search

The choice is a backtrack point! (May have to backtrack, try to achieve ¬garbage by *roll* instead)

g = {clean, ¬garbage, served, wrapped}

Solution Extraction (4)



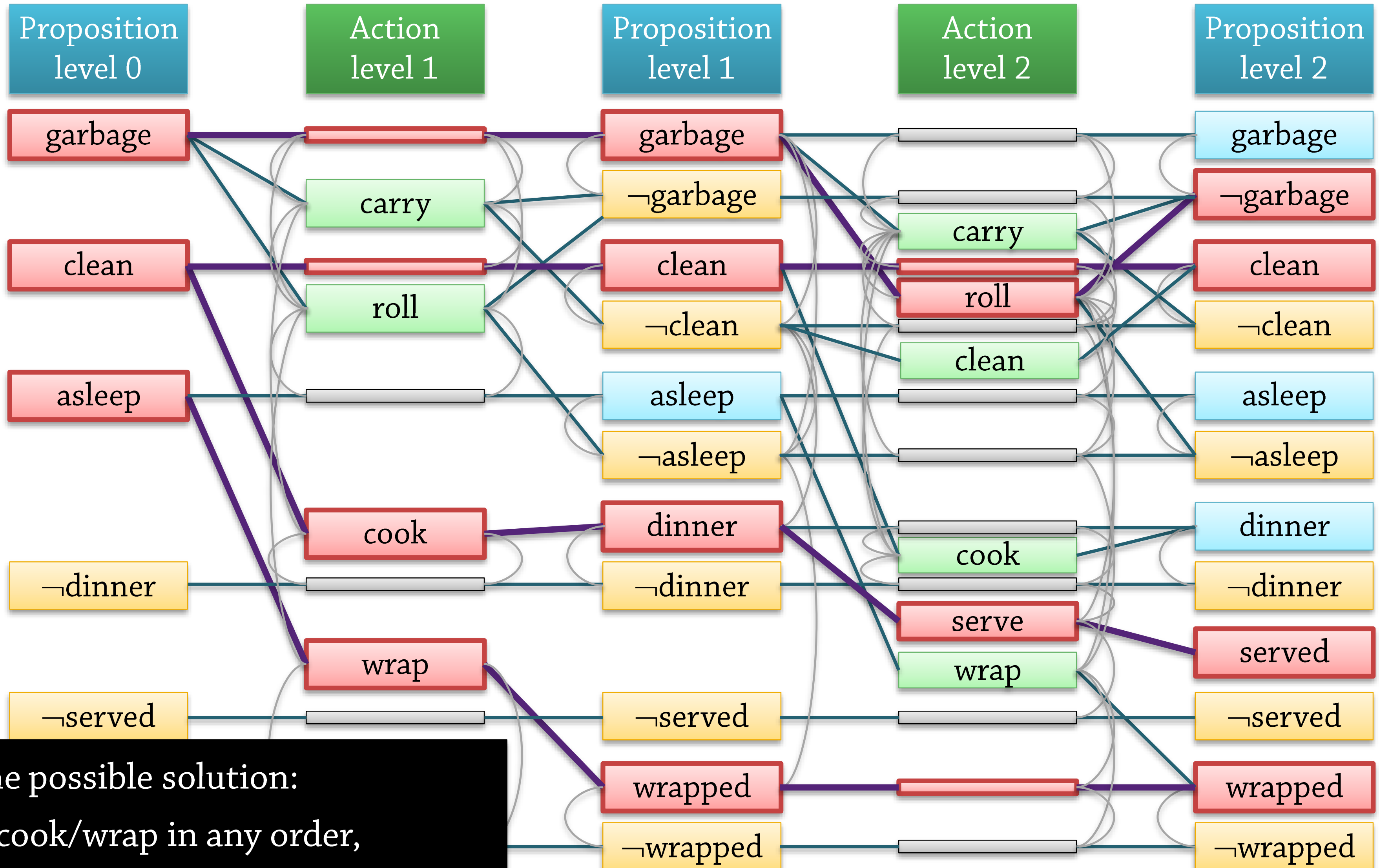
The successor has a new goal

This goal consists of **all** preconditions of the chosen action (*Not* a backtrack point)

The goal is not mutually exclusive (if so, it would be unreachable)

Successor goal = {clean, **garbage**, **dinner**, wrapped}

Solution Extraction (5)



One possible solution:

1) cook/wrap in any order,

2) serve/roll in any order

Solution Extraction (6)

The set of goals we are trying to achieve

The proposition level, starting at the highest level

procedure **Solution-extraction**(*g*,*i*)

<u>if</u> *i*=0 then return the solution

<u>nondeterministically choose</u>

a **set** of **non-mutex** actions
("real" actions and/or maintenance actions)
to use in state s_{i-1} to achieve *g*
(must achieve the *entire* goal!)

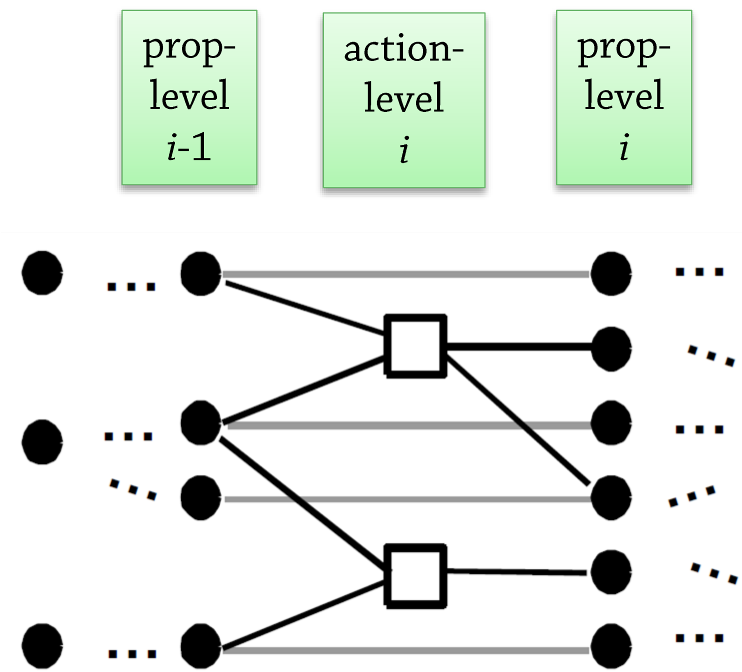
<u>if</u> no such set exists then fail (backtrack)

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

Solution-extraction(*g'*, *i*–1)

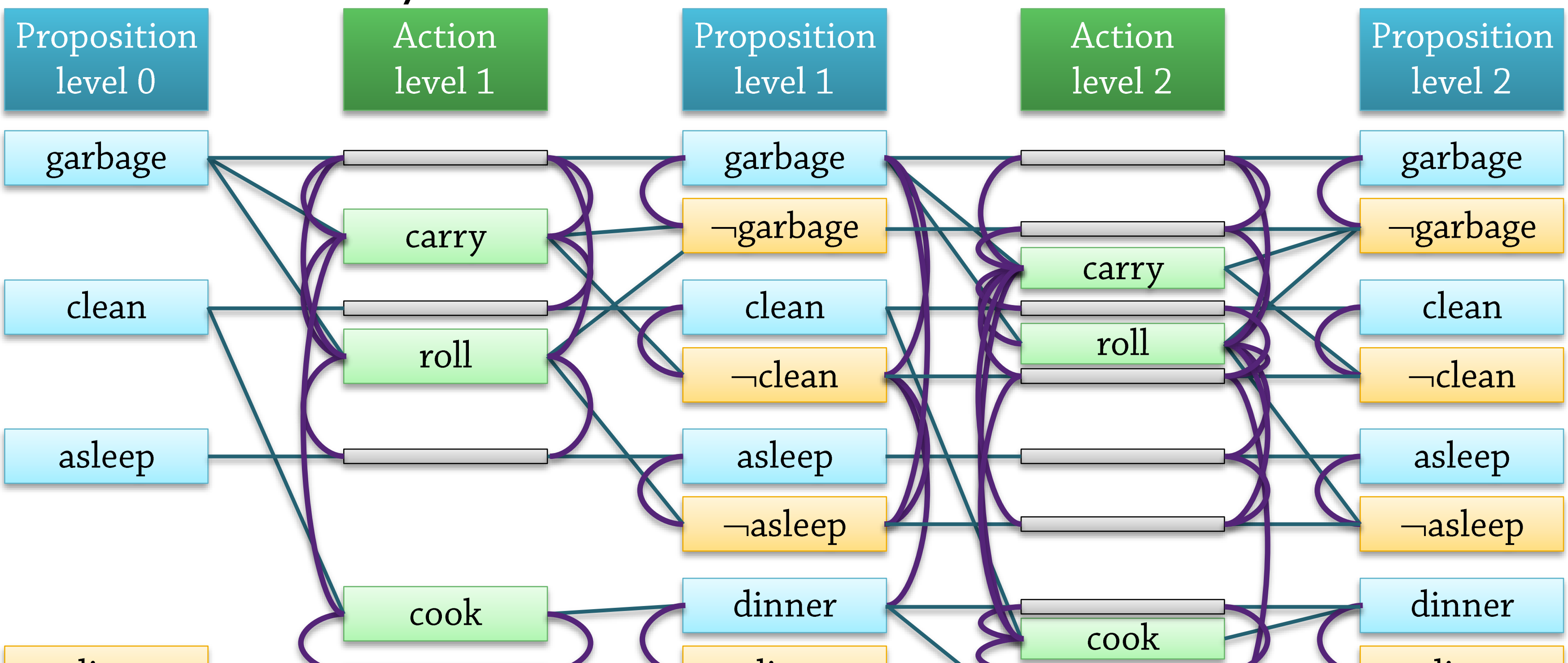
end Solution-extraction

A form of **<u>backwards search</u>**, but <u>only</u> among the actions in the graph (generally much fewer, esp. with mutexes)!



Important Properties

- **Possible literals**:
  - What is achieved is always carried forward by no-ops
  - ➔ Monotonically increase over proposition levels
- **Possible actions**:
  - Action included if all precondition literals exist in the preceding prop level
  - ➔ Monotonically increase over action levels



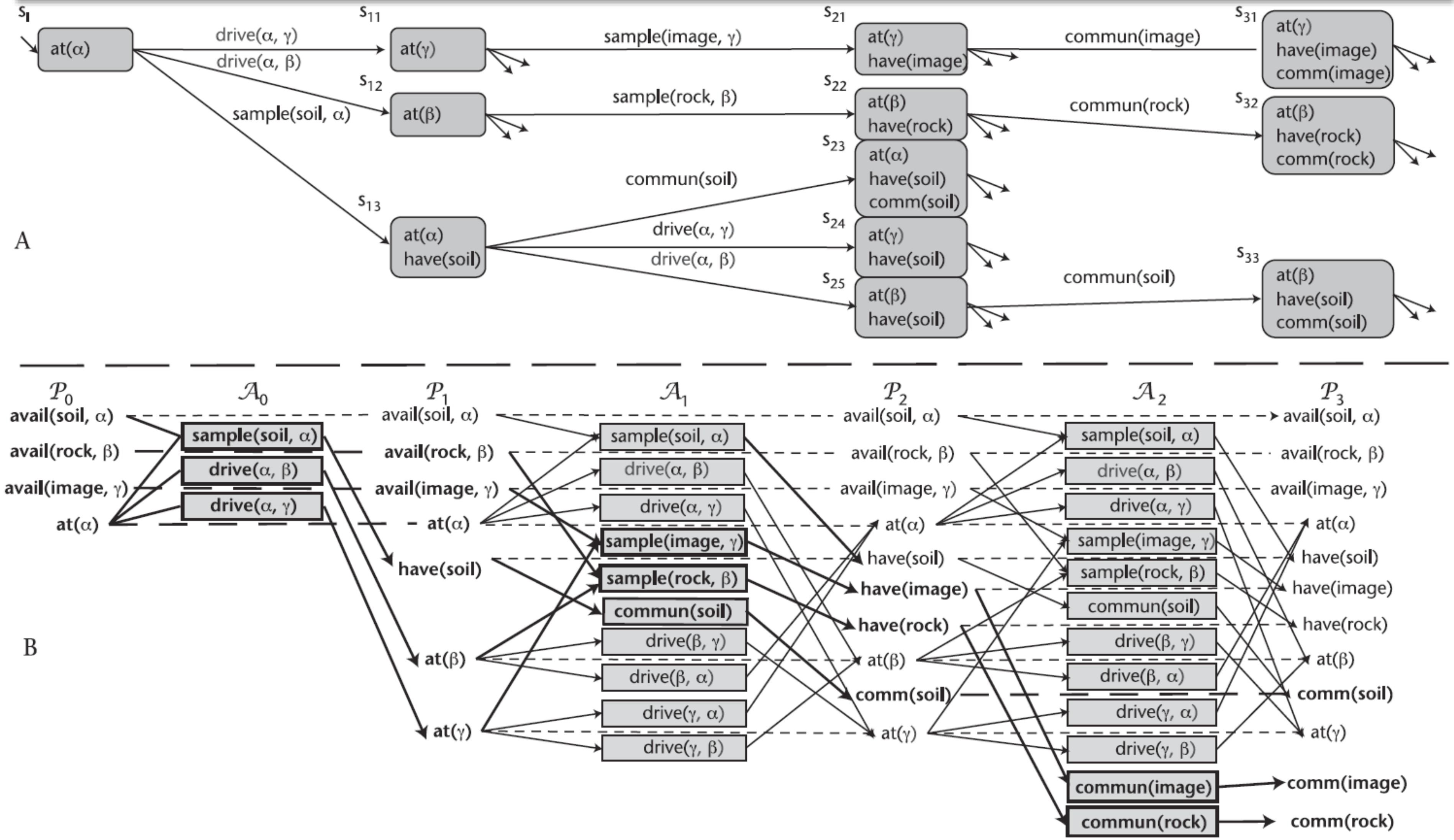
Important Properties (2)

- Mutex relationships:
  - Mutexes between *included literals* monotonically decrease
    - If two literals could be achieved together in the previous level, we could always just "do nothing", preserving them to the next level
  - (Mutexes to *newly added* literals can be introduced)

- At some point, the Planning Graph "levels off"
  - After some time k all levels are identical
  - (Why?)

A **planning graph** is **exactly** what we described here – **not** some arbitrary planning-related graph

Top: Real state reachability for a Rover problem ("avail" facts not shown)



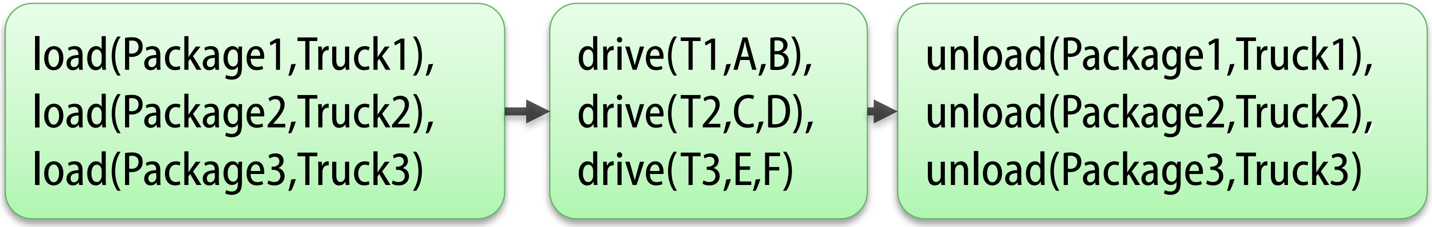
Bottom: Planning Graph for the same problem (mutexes not shown)

At each step, an *overestimate* of reachable propositions / applicable actions!

Parallel Optimality

- A form of iterative deepening:

- Therefore, GraphPlan is *optimal in the number of time steps*
  - Not perfect, as we normally care much more about:
    - Total action cost
    - Number of actions (special case where action cost = 1)
    - Total execution time ("makespan")



Relaxed Planning Graph Heuristics

Heuristics as approximations of h+ (optimal DR)

$h_1(s) <= h+(s)$

cost(p and q) = **max**(cost(p), cost(q))

<u>Optimistic relative to h^+</u>:
As if achieving the *most expensive* goal would *always* achieve all the others

Gives far too little information

$h_0(s) = h_{add}(s) >= h+(s)$

cost(p and q) = **sum**(cost(p), cost(q))

<u>Pessimistic relative to h^+</u>:
As if achieving one goal could *never* help in achieving any other

Informative, but always exceeds h+ and can exceed even h* by a large margin!

How can we take some <u>interactions</u> into account?

Relaxed Planning Graphs

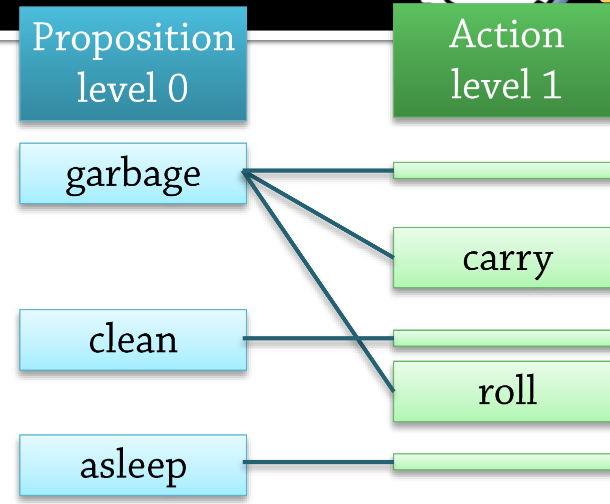
- The **planning graph** takes many interactions into account
  - One possible heuristic h(s):
    - Use GraphPlan to find a solution starting in s
    - Return the number of actions in the solution
  - Too slow (requires plan generation), *but*:

Let's apply **delete relaxation**, *then* construct a planning graph!

(Called a *relaxed planning graph* -- pioneered by FastForward, FF)

Recall: Delete relaxation assumes we have **positive preconds, goals**

Building Relaxed Planning Graphs



- Building a relaxed planning graph:
  - Construct **proposition level 0** (PL0)
    - Atoms in the initial state
  - Construct **action level 1** (AL1)
    - Actions whose preconditions are included in PL0
  - Two actions in AL1 are **mutex** if:
    - Their effects are inconsistent
    - One destroys a precondition of the other

Can't happen!
Only positive effects!

- Construct **state level 1** (PL1)
  - All effects of actions in AL1
- Two propositions in PL1 are **mutex** if:
  - One is the negation of another
  - All actions that achieve them are mutex

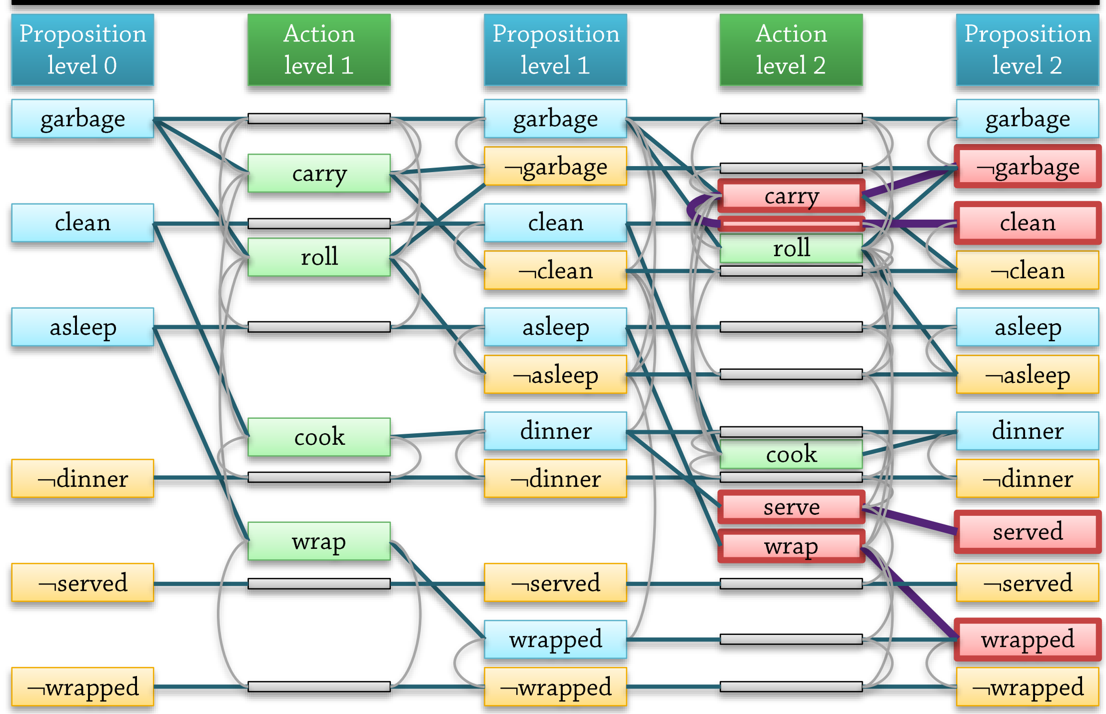
Can't happen!
Only positive propositions,
no mutex actions!

No Backtracking

- **No backtracking**: Recall the backwards search procedure
  - Goal specifies which propositions to achieve in **PL2**
  - Choose one of many possible sets of achieving actions in **AL2**
    - *If they are mutex, backtrack*
  - Determine which propositions must be achieved in **PL1** (preconds of actions)
  - Choose actions in **AL1**
    - *If they are mutex, backtrack*
  - …

No delete effects
➔
No mutexes
➔
No backtracking

Situation *with* delete effects:



Properties of Relaxed Planning Graphs

- The **relaxed planning graph** considers **positive** interactions
  - For example, when one action achieves multiple goals
  - Ignores **negative** interactions
  - No delete effects → **no mutexes** to calculate (no inconsistent effects, no interference, …)
  - No mutexes exist → can select **more actions** per level, **fewer levels** required
  - No mutexes exist → **no backtracking** needed in solution extraction
  - Can extract a **Graphplan-optimal** relaxed plan (minimal number of steps) in **polynomial** time

$h_{FF}(s)$ = number of actions in relaxed plan from state s

Complexity

How can this be efficient?
Sounds as if we calculate h+, which is **NP-complete**!

- The plan that is extracted is only *GraphPlan*-optimal!
  - Optimal number of time steps
  - Possibly sub-optimal number of *actions* (or suboptimal *action costs*)
  - ➔ h_{FF} is **not admissible**,
    can be *greater than* h+ (but not smaller!)
    and can be *greater than* h* (or smaller)

- Still, the delete-relaxed plan *can* take positive interactions into account
  - ➔ Often closer to true costs than h_{add} is

- Plan extraction can use several heuristics (!)
  - Trying to reduce the *sequential* length of the relaxed plan,
    to get even closer to true costs

FastForward's Search Strategy: Enforced Hill-Climbing

HSP

- Recall hill climbing in HSP!
  - Works **approximately** like this (some intricacies omitted):
    - ```
      impasses = 0;
      unexpanded = { };
      current = initialNode;
      while (not yet reached the goal) {
            children ← expand(current);          // Apply all applicable actions
            if (children = ∅) {
                  current = pop(unexpanded);
            } else {
                  bestChild ← best(children);
                  add other children to unexpanded in order of h(n);
                  if (h(bestChild) ≥ h(current)) {
                        impasses++;
                        if (impasses == threshold) {
                              current = pop(unexpanded);
                              impasses = 0;
                        }
                  }
      }     }
      ```

At each step, choose a child with minimal heuristic value

Allow a few steps without improvement of heuristic value

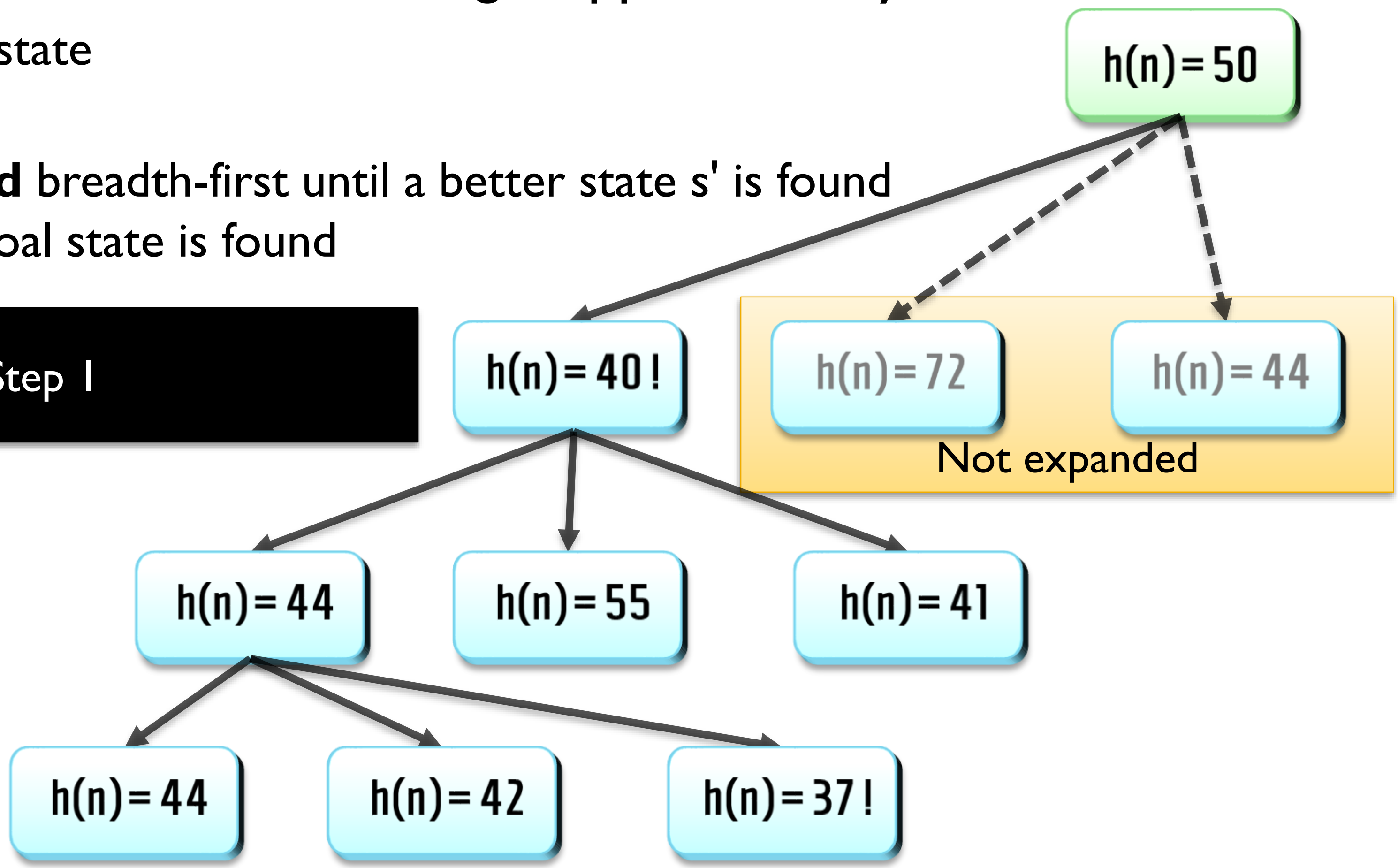
Too many such steps → Restart at some other point
      ```

Enforced Hill Climbing

- FF uses **enforced** hill climbing – approximately:
  - s ← init-state
  - **repeat**
    expand breadth-first until a better state s' is found
    until a goal state is found

Step 1

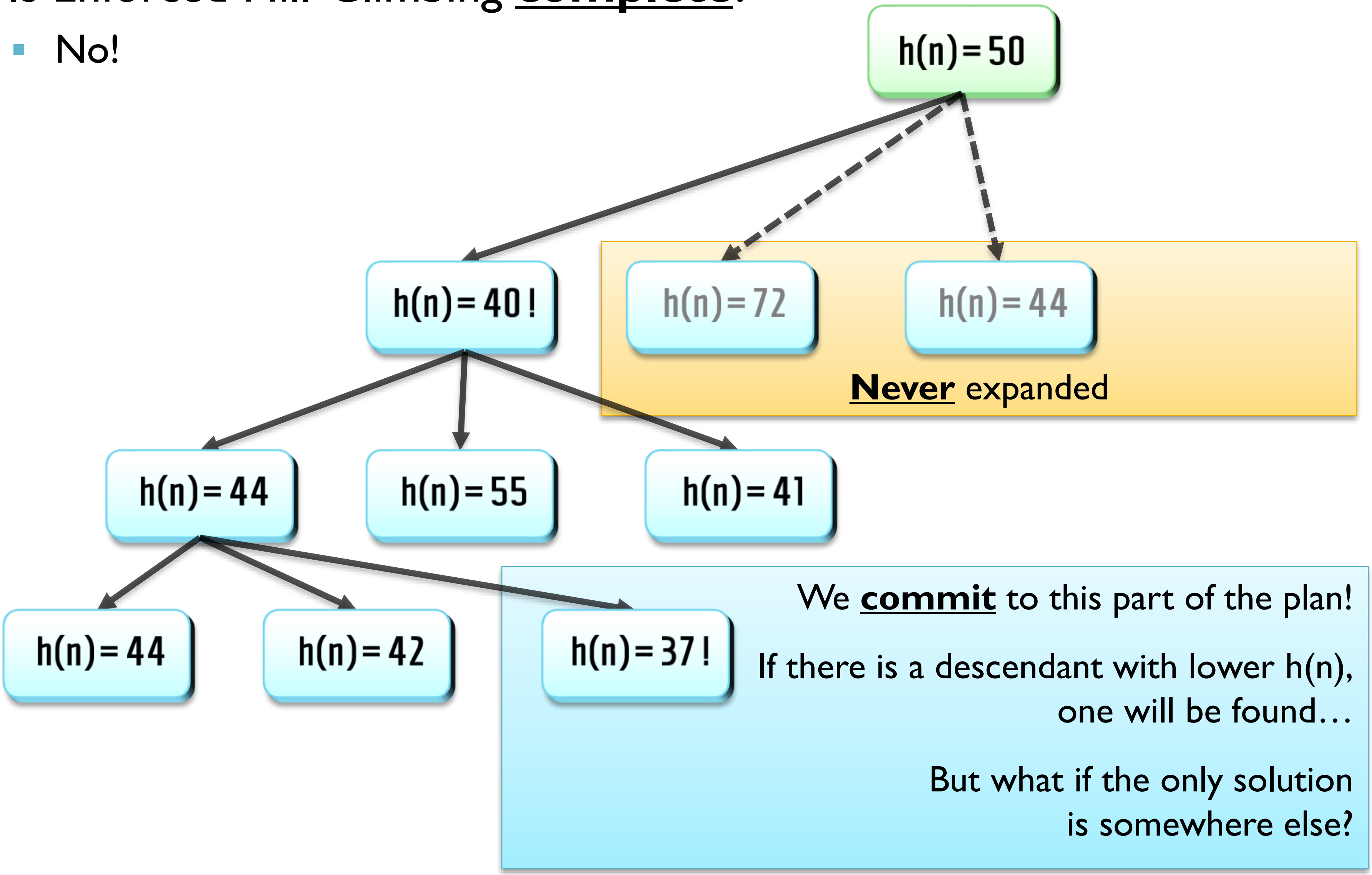
Step 2



Wait longer to decide which branch to take
Don't restart – keep going

Properties of EHC

- Is Enforced Hill-Climbing **complete**?
  - No!



Properties of EHC

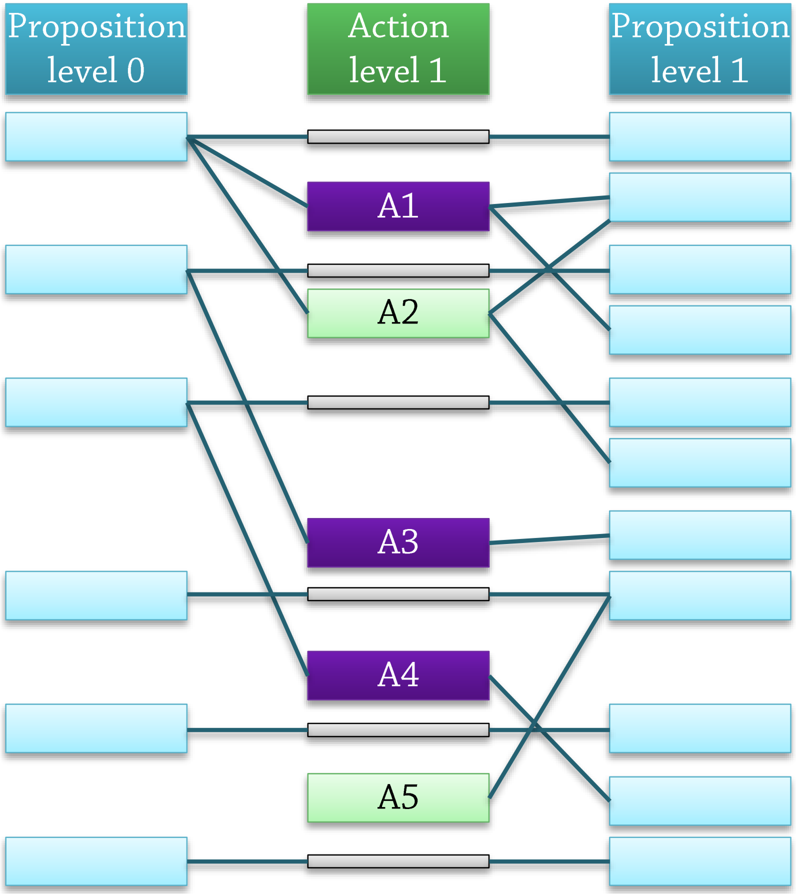
- If you reach a **dead end**:
  - HSP used *random restarts*
  - FF uses *best-first search* from the initial state (with an *open* list), using *only* the inadmissible FF heuristic for guidance (no consideration of "cost so far")
    - So **FF** is complete, but EHC is not

- Is Enforced Hill-Climbing **efficient / effective**?
  - In many cases (when paired with FF's Relaxed Planning Graph heuristic)
  - But can spend considerable time on:
    - Breadth first search to escape plateaus / local minima
    - Best first search when EHC does not work
    - Analysis: Hoffmann (2005), *Where `ignoring delete lists' works: Local search topology in planning benchmarks*

"Helpful Actions" in FF

Helpful Actions

- Pruning Technique in FF: **Helpful Actions** in state *s*
  - Recall: FF's heuristic function for state *s*
    - Construct a *relaxed planning graph* starting in *s*
    - Extract a *relaxed plan* (choose actions among potentially executable in each level)
  - **Helpful actions**:
    - The actions selected in action level 1
    - Plus all *other* actions that could achieve the *same subgoals* (but did not happen to be chosen)
    - More likely to be useful to the plan than other *executable* actions
  - FF constrains EHC to only use helpful actions!
    - Sometimes also called *preferred operators*



FF: Helpful Actions (2)

Suppose this is the relaxed plan generated by FF…

Delete-relaxed:

s_0 = {clean, garbage, asleep}

g = {clean, served, wrapped}

Action	Preconds	Effects
cook()	clean	dinner
serve()	dinner	served
wrap()	asleep	wrapped
carry()	garbage	
roll()	garbage	
clean()	clean	

Possible in A-level 1:
carry, roll, clean, cook, wrap

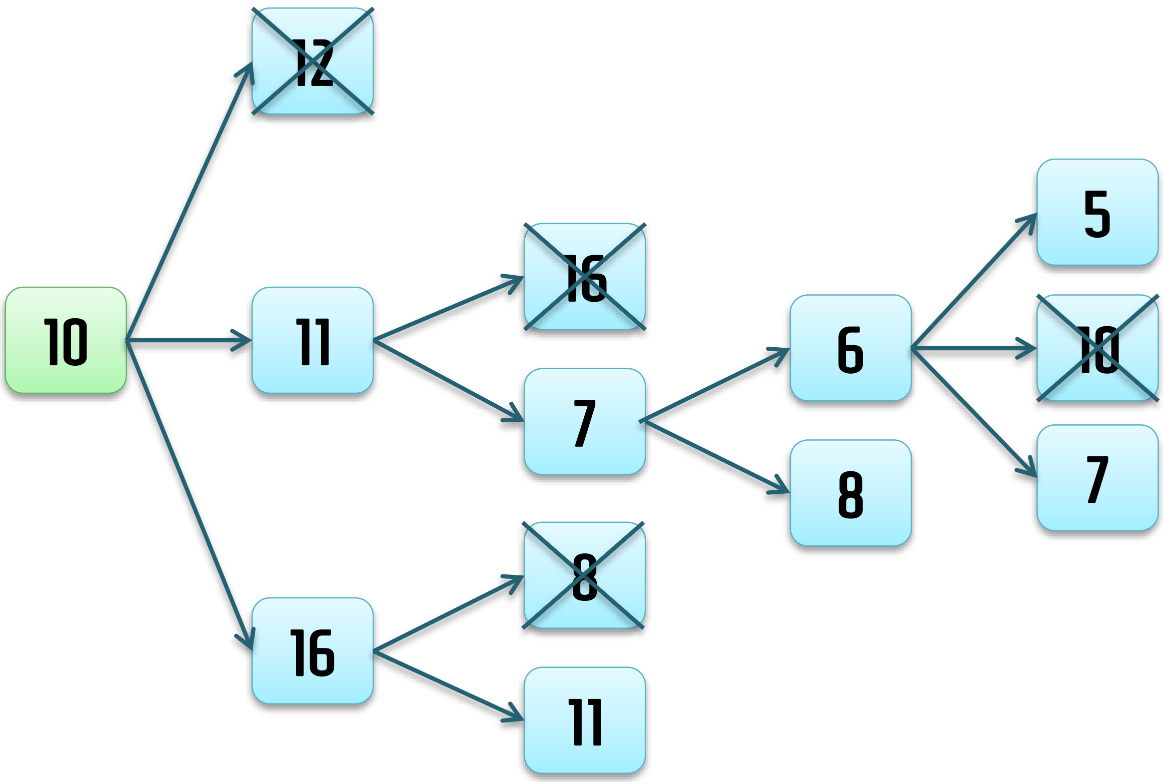
Used in relaxed plan:
cook, wrap

Achieved at P-level 1:
clean, dinner, wrapped

Helpful actions:
cook, wrap, clean

FF: EHC with Helpful Actions

- **EHC** with **helpful actions**:
  - Non-helpful actions crossed over, never expanded



FF: EHC with Helpful Actions (2)

- **EHC** with **helpful actions**:
  - EHC(initial state I , goal G)

plan ← EMPTY
s ← I
while $h_{FF}(s)$!= 0 do
    execute **breadth first** search from s,
    using *only helpful actions*,
    to find the first s' such that $h_{FF}(s') < h_{FF}(s)$

    if no such state is found **then fail**

    plan ← plan + actions on the path to s'
    s ← s'
end while
return plan

Incomplete
if there are dead ends!

If EHC fails, fall back on best-first search using $f(s)=h_{FF}(s)$