Automated Planning

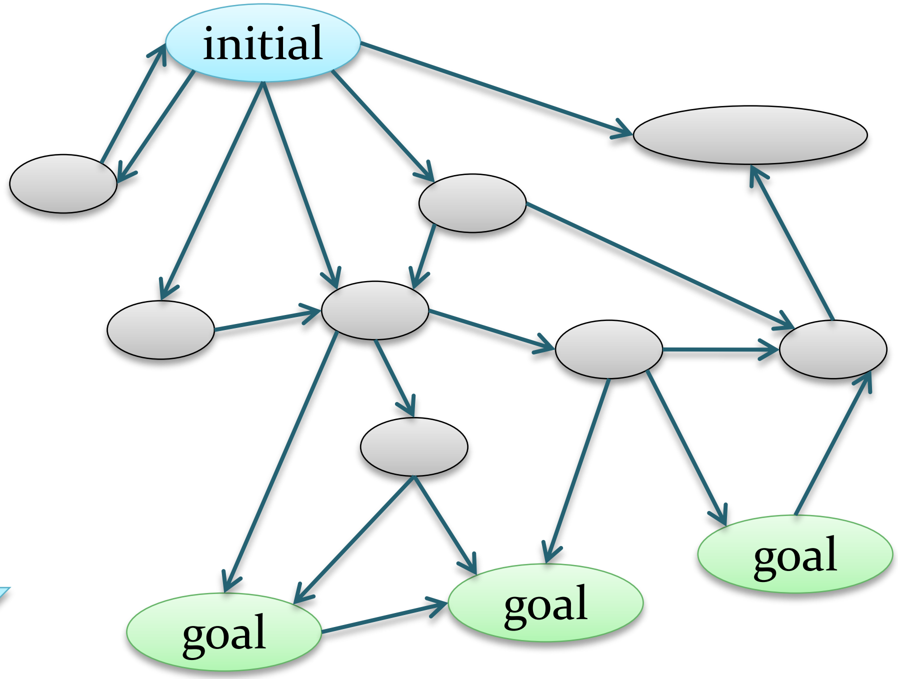
Backward State Space Search

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

Forward Search (repetition)

- Classical Planning: **<u>Find a path in a finite graph</u>**

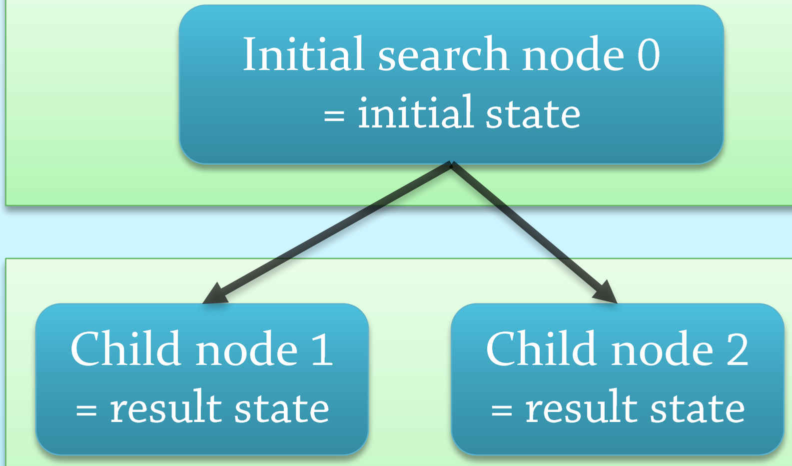
We've seen this direction:
Forward search,
initial ➔ goal



Forward State Space (repetition)

Forward State Space

Forward planning, forward-chaining, progression: Begin in the **initial state**



Initial search node 0
= initial state

Child node 1
= result state

Child node 2
= result state

Corresponds directly to the initial state

Edges correspond to applicable actions

Given a state s:

For every action *a* **applicable** in *s*, generate **the state** that results from **applying** *a* to *s*

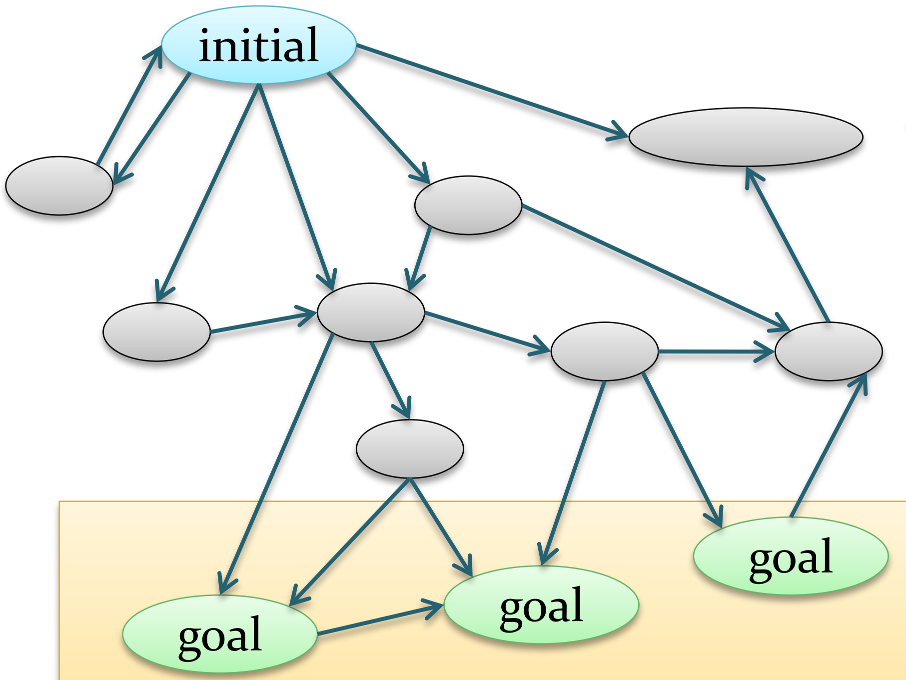
How to interpret a node:
If I can find a way
from this state to a goal state,
then I have a complete plan

Goal criterion: *The state of the node satisfies the goal formula*

Plan extraction: *Generate the sequence of all actions on the path to the goal node*

Forward and Backward Search

- Classical Planning: **<u>Find a path in a finite graph</u>**



<u>A complication in backward search</u>:
We have *multiple* goal states

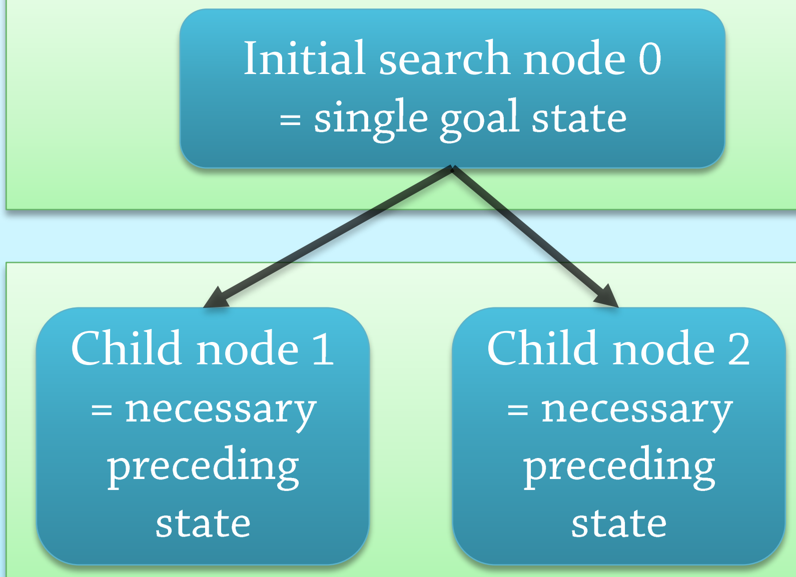
What about
this direction:
Backward,
goal ➔ initial?

"Simple" Backward State Space

First, the **simple case**: A single goal state!

"Simple" Backward State Space

Backward planning, backward-chaining, regression: Begin in the **goal state**



Corresponds to the *single goal state*

Edges correspond to
"*relevant* actions, executed *backwards*"

Given a goal state g:

For every action *a* **relevant** to g,
generate **the state**
in which executing *a* **would result in** g

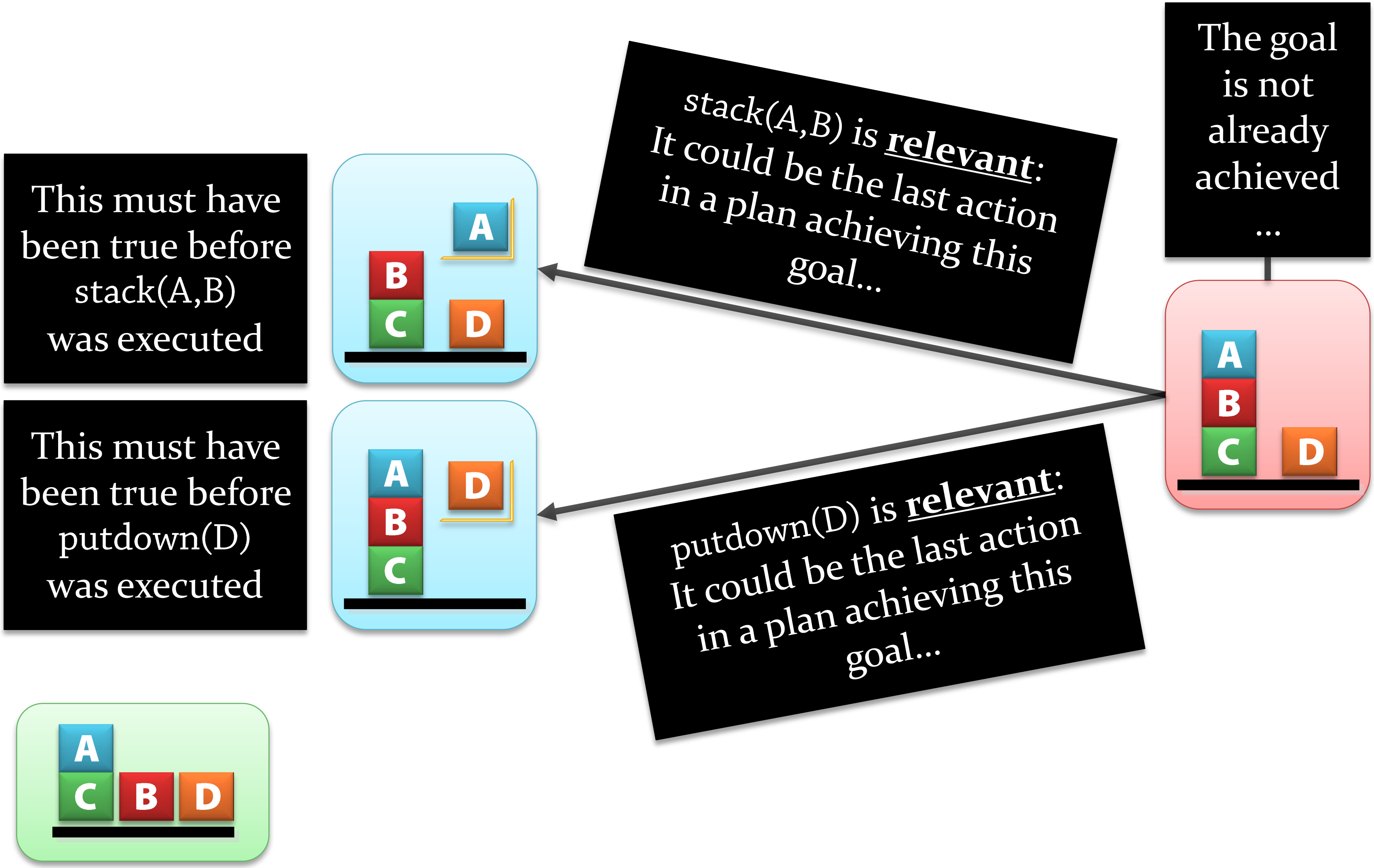
How to interpret a node:
If I can find a way
from the initial state to this goal state,
then I have a complete plan

Stop criterion: *The initial state satisfies the current goal state*

Plan extraction: *Generate the sequence of all actions on the constructed path*

“Simple” Backward Search (1)

”Simple” case: A single goal state!



The goal is not already achieved …

stack(A,B) is **relevant**: It could be the last action in a plan achieving this goal…

This must have been true before stack(A,B) was executed

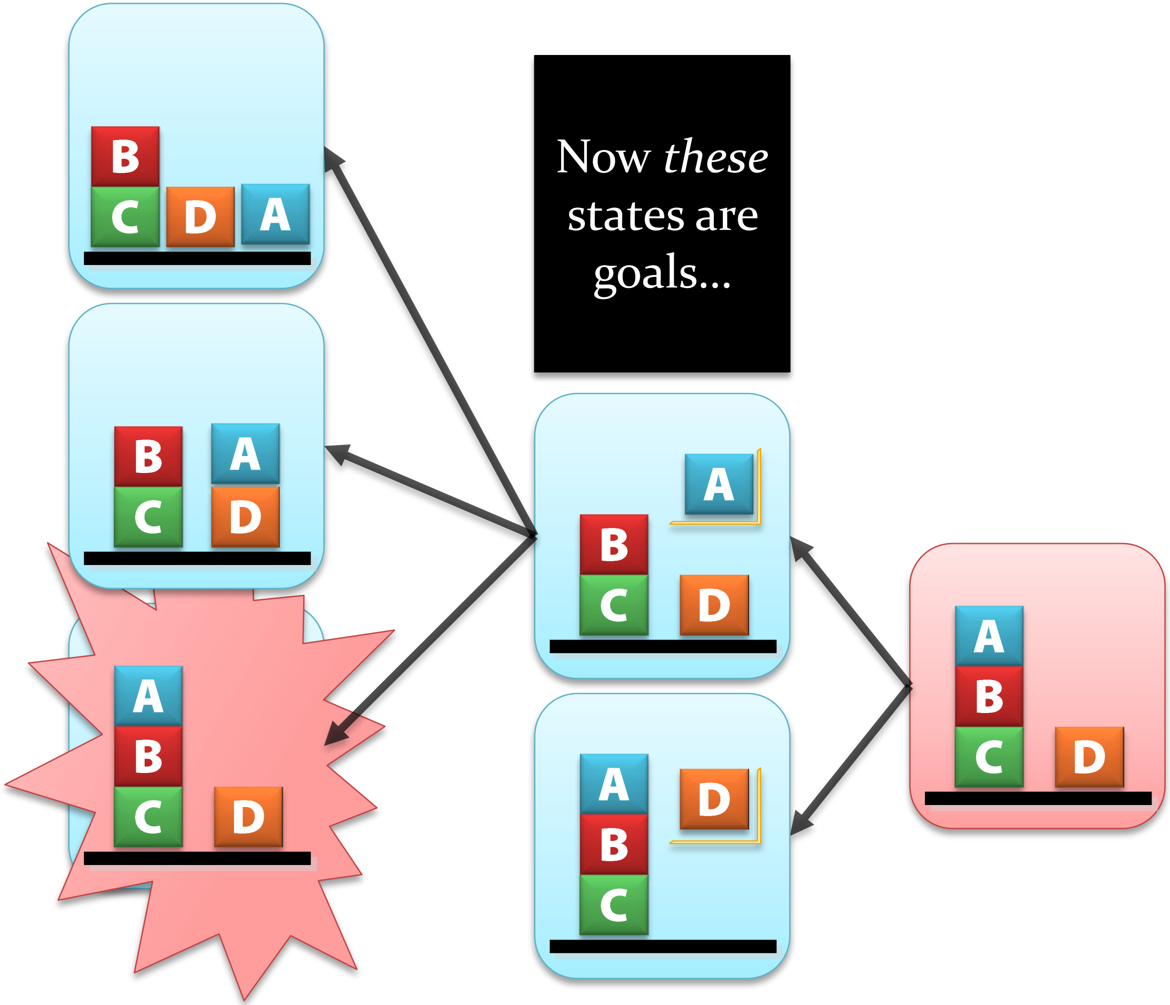
putdown(D) is **relevant**: It could be the last action in a plan achieving this goal…

This must have been true before putdown(D) was executed

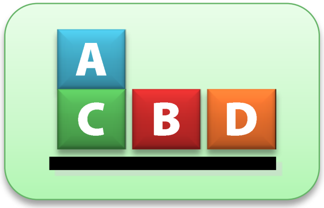
"Simple" Backward Search (2)

"Simple" case: A single goal state!

- Relevant actions:
  - pickup(A)
  - unstack(A,D)
  - unstack(A,B)
    - Cycle!



Now *these* states are goals…

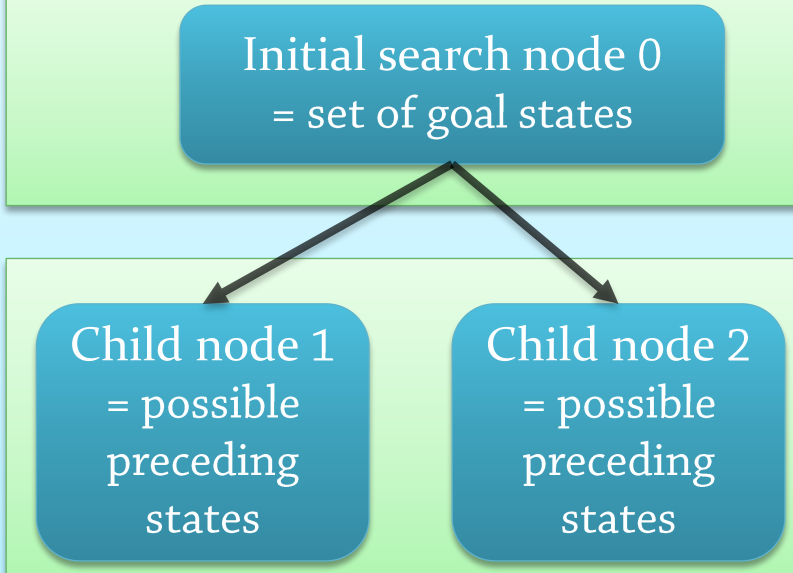


"Complete" Backward State Space

Second, allow **sets of goal states**...

Backward State Space

Backward planning, backward-chaining, regression: Begin in the **set of goal states**



Corresponds to the *set of goal states*

Edges correspond to
"*relevant* actions, executed *backwards*"

Given a set G of goal states:

For every action *a* **relevant** to G,
generate the **set of states**
from which *a* **would result in** a state in G

How to interpret a node:
If I can find a way from the initial state
to **one of these states**,
then I have a complete plan

Stop criterion:
The initial state satisfies
***at least one** of the node's goal states*

Plan extraction: *Generate the sequence of all actions on the constructed path*

State Sets and Regression

How do we represent a set of goal states?

Arbitrary set of states?

Too expensive to calculate
regression, relevant actions

Use classical representation!

Arbitrary set of *ground goal literals:*
$g = \{ in(c1,p3), \ldots, in(c5,p3), \neg foo \}$

$\gamma(s,a)$ = the state resulting from executing a in state s

$\Gamma(s) = \{ \gamma(s,a) \mid a \in A$ and a is applicable to s $\}$

Forward / progression:
Which state do I end up in?

$\gamma^{-1}(g,a) = \{ s \mid$ a is applicable to s and $\gamma(s,a)$ satisfies g $\}$
$= (g - effects(a)) \cup precond(a)$

Backward / regression:
Which states could I start from?

Everything except
effects(a) must
already be true before a

precond(a) was true,
so a was applicable

$\Gamma^{-1}(g) = \{ \gamma^{-1}(g,a) \mid a \in A$ and a is relevant for g $\}$

All successor *subgoals*

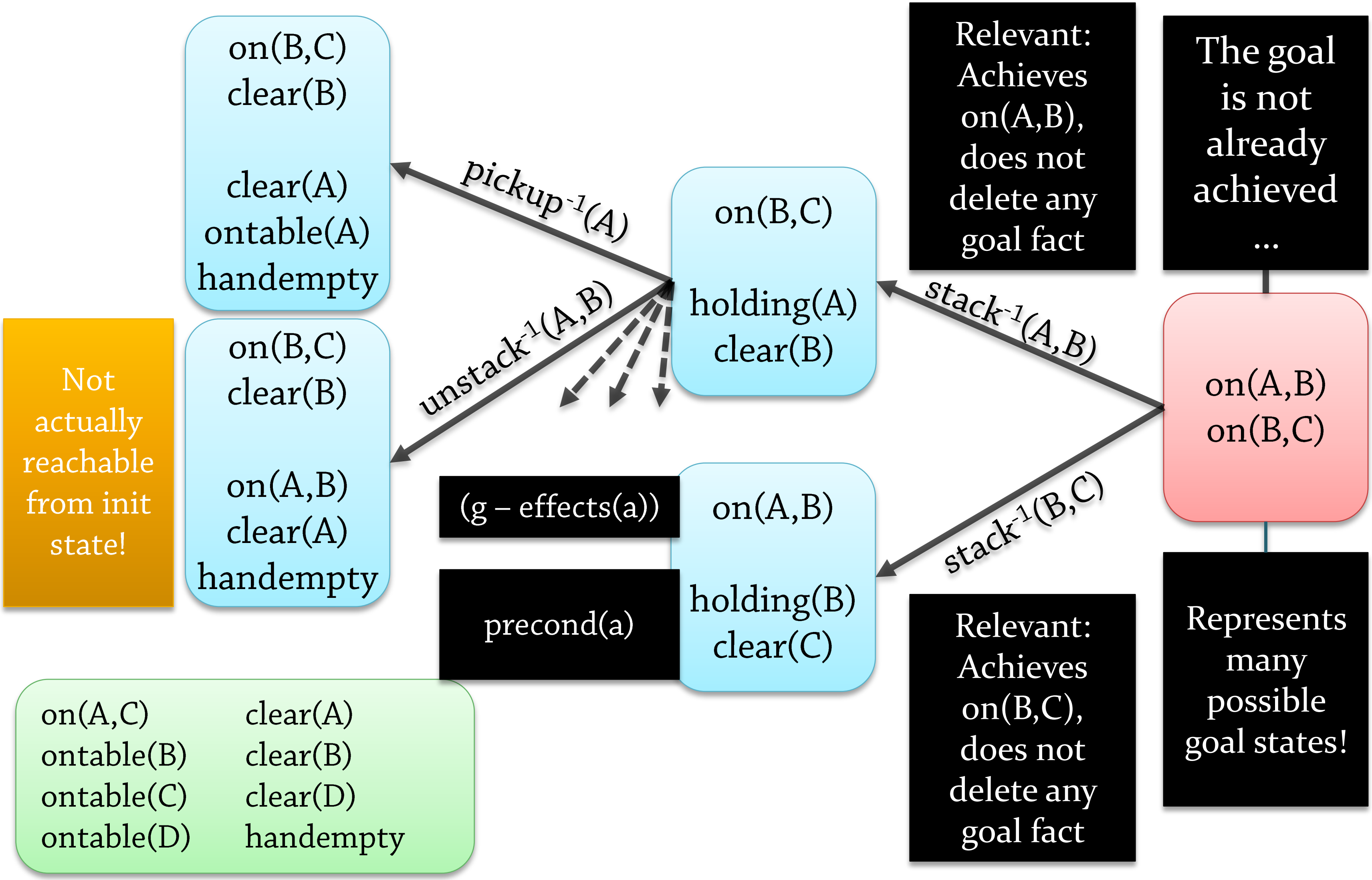
a is **relevant** for g iff

$g \cap effects(a) \neq \emptyset$ and
$g+ \cap effects-(a) = \emptyset$ and
$g- \cap effects+(a) = \emptyset$

Contribute to the goal,
and do not **destroy** it

Regression

Regression Example



on(A,B)
on(B,C)

The goal is not already achieved …

Represents many possible goal states!

stack^{-1}(A,B)

Relevant: Achieves on(A,B), does not delete any goal fact

on(B,C)
holding(A)
clear(B)

stack^{-1}(B,C)

Relevant: Achieves on(B,C), does not delete any goal fact

on(A,B)
holding(B)
clear(C)

(g – effects(a))

precond(a)

pickup^{-1}(A)

on(B,C)
clear(B)
clear(A)
ontable(A)
handempty

unstack^{-1}(A,B)

on(B,C)
clear(B)
on(A,B)
clear(A)
handempty

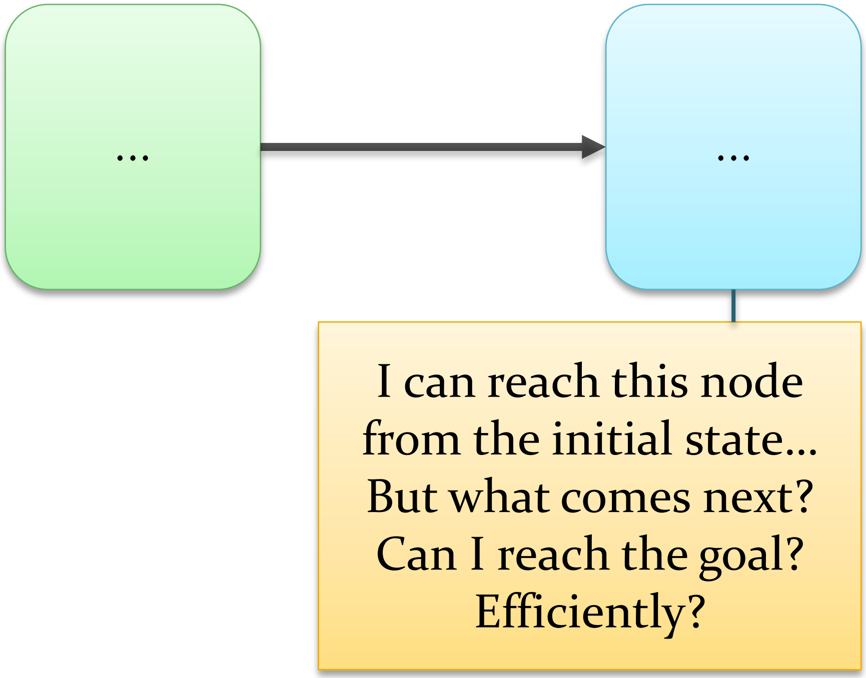
Not actually reachable from init state!

on(A,C)
ontable(B)
ontable(C)
ontable(D)
clear(A)
clear(B)
clear(D)
handempty

Symmetric Problems

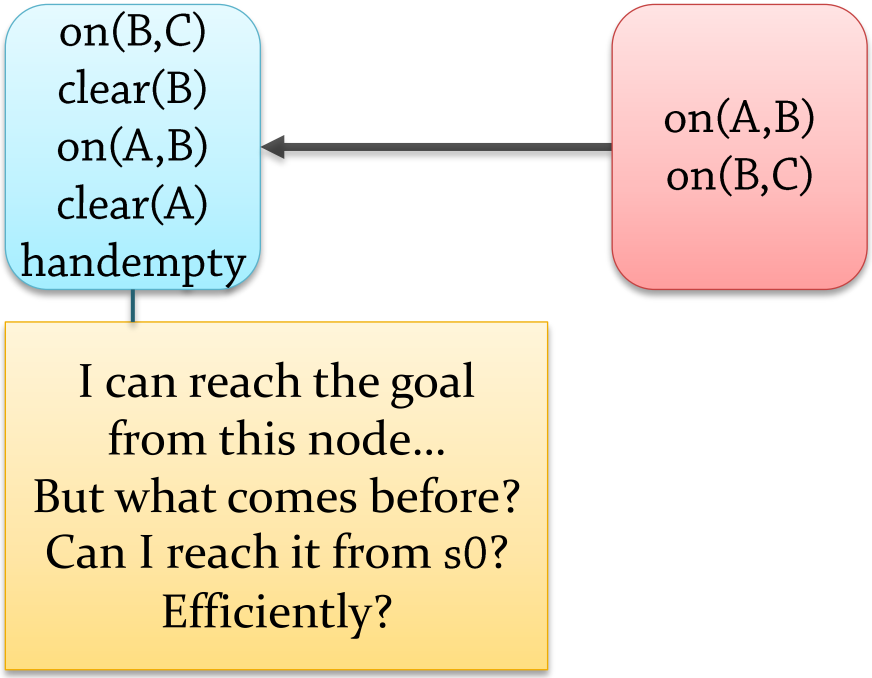
Symmetric problems!

Forward search



I can reach this node from the initial state…
But what comes next?
Can I reach the goal?
Efficiently?

Backward search



I can reach the goal from this node…
But what comes before?
Can I reach it from s0?
Efficiently?

Backward and Forward Search

FORWARD SEARCH

- Problematic when:
  - There are many **<u>applicable</u>** actions
    → high branching factor
    → need guidance
  - Blind search knows
    if an action is applicable,
    but not if it will contribute
    to the goal

BACKWARD SEARCH

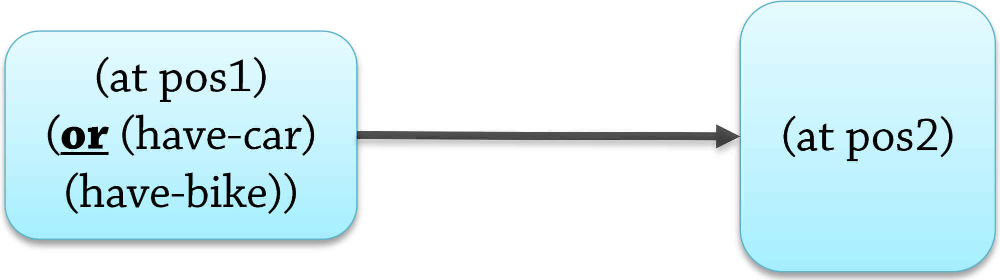
- Problematic when:
  - There are many **<u>relevant</u>** actions
    → high branching factor
    → need guidance
  - Blind search knows
    if an action contributes to the goal,
    but not if you can achieve its
    preconditions

Blind backward search
is **<u>generally</u>** better than blind forward search:
Relevance **<u>tends</u>** to provide better guidance than applicability

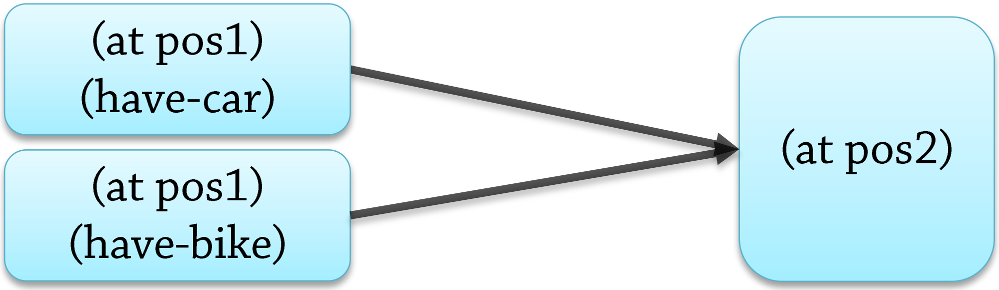
But this **<u>in itself</u>** is **<u>not</u>** enough to generate plans quickly!

Backward Search and Expressivity

- Let's take a look at expressivity:
  - Suppose we have **disjunctive preconditions**
    - (:**action** travel
      :**parameters** (?from ?to – location)
      :**precondition** (**or** (have-car) (have-bike))
      :**effects** (**and** (at ?to) (not (at ?from))))
  - How do we apply such actions backwards?
    - More complicated **disjunctive goals** to achieve?



    - Additional **branching**?



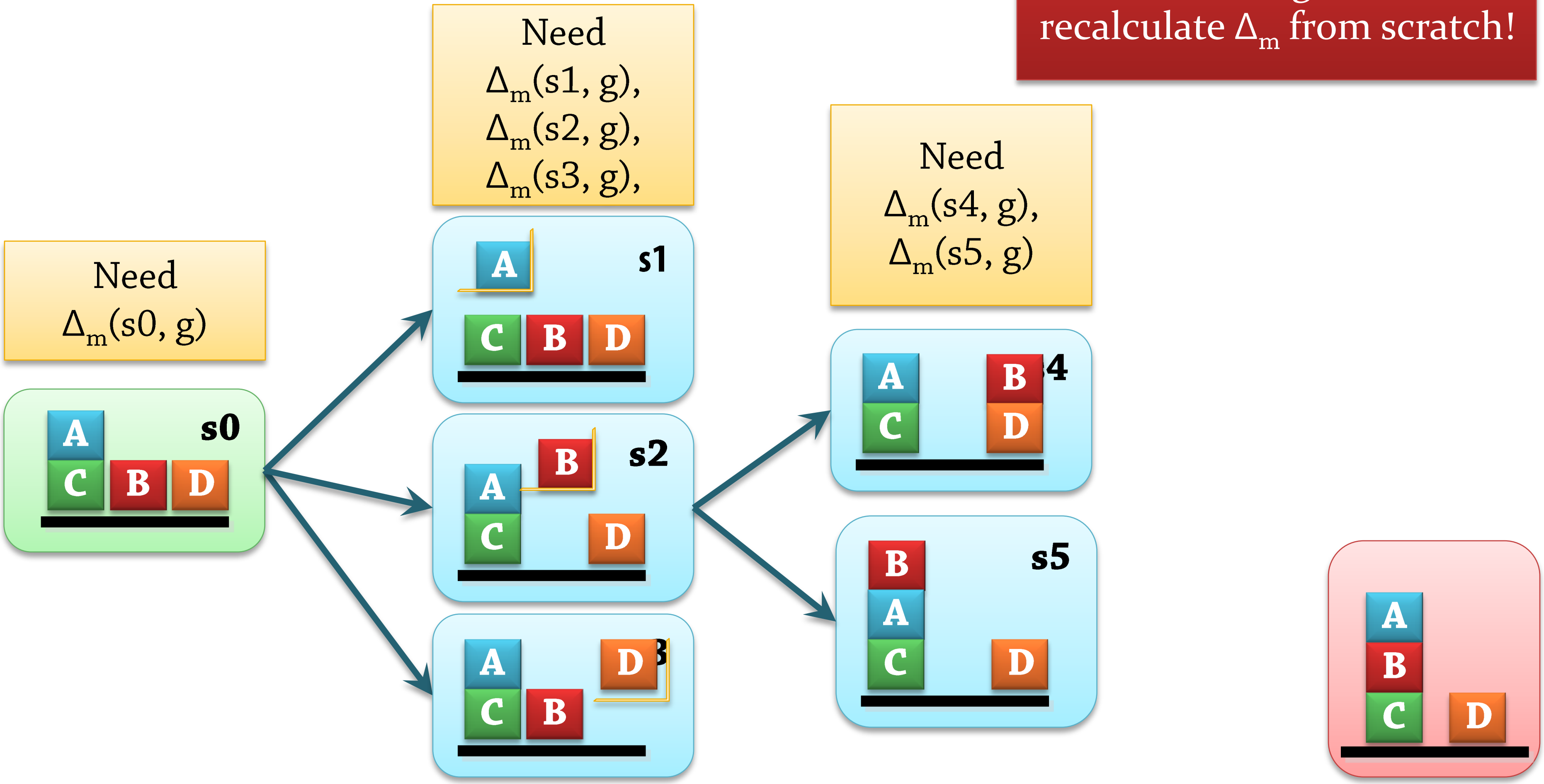
Some extensions are less straight-forward in backward search (but possible!)

Backward Search and Heuristics

Forward Search with h_m

- Consider h_m heuristics using forward search:

New search node → new starting state → recalculate Δ_m from scratch!



Need
$\Delta_m(s0, g)$

Need
$\Delta_m(s1, g)$,
$\Delta_m(s2, g)$,
$\Delta_m(s3, g)$,

Need
$\Delta_m(s4, g)$,
$\Delta_m(s5, g)$

Backward Search with h_m

- What about **backward** search?

Need
$\Delta_m(s0, g3)$,
$\Delta_m(s0, g4)$,
$\Delta_m(s0, g5)$

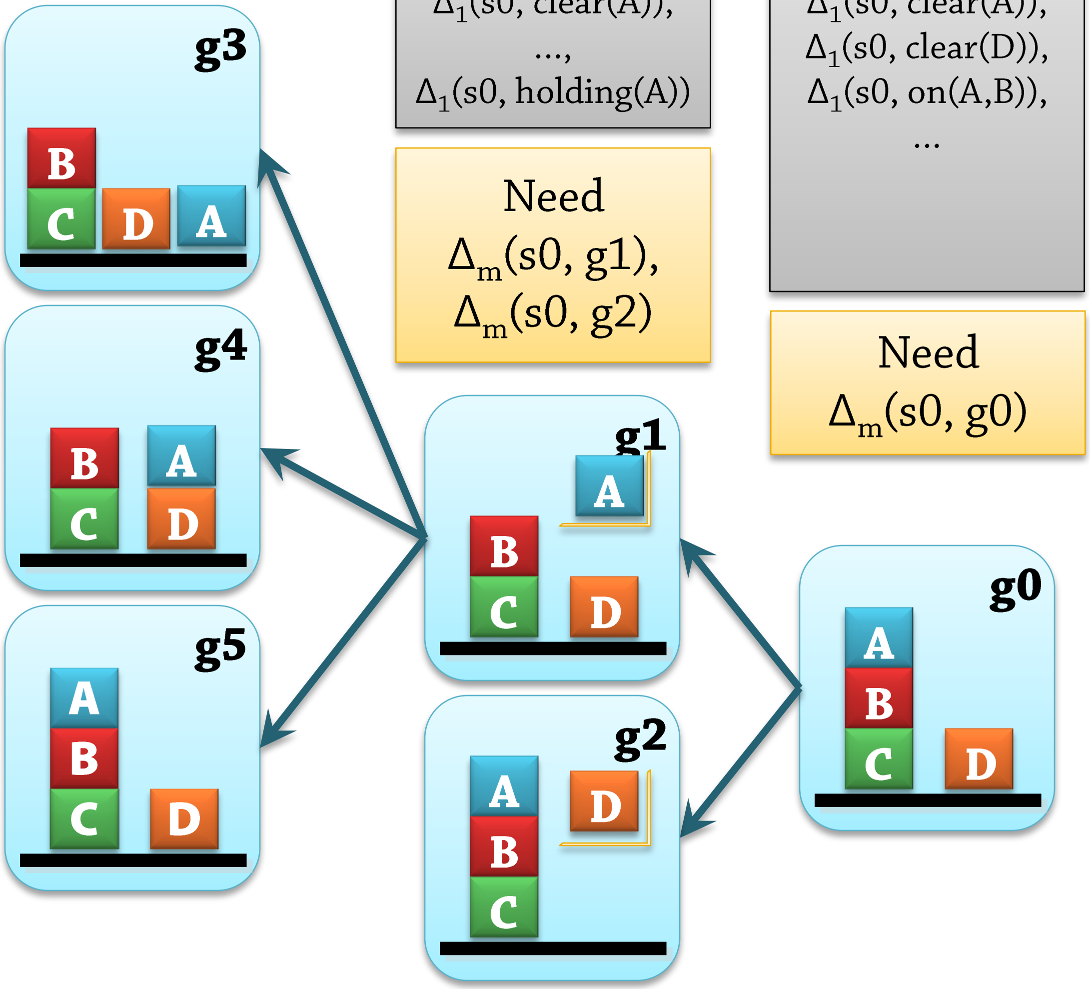
New search node →
same starting state →
use the old Δ_m values
for those goal subsets
that were already
calculated!

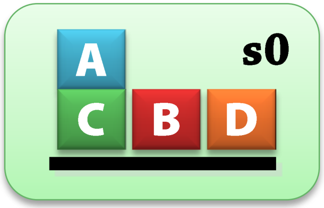
$\Delta_1(s0, g1)$
depends on
$\Delta_1(s0, ontable(C))$,
$\Delta_1(s0, ontable(D))$,
$\Delta_1(s0, clear(A))$,
…,
$\Delta_1(s0, holding(A))$

Need
$\Delta_m(s0, g1)$,
$\Delta_m(s0, g2)$

$\Delta_1(s0, g0)$
depends on
$\Delta_1(s0, ontable(C))$,
$\Delta_1(s0, ontable(D))$,
$\Delta_1(s0, clear(A))$,
$\Delta_1(s0, clear(D))$,
$\Delta_1(s0, on(A,B))$,
…

Need
$\Delta_m(s0, g0)$





HSPr, HSPr*

- Results:
  - Faster calculation of heuristics
  - Applied in HSPr (non-optimal) and HSPr* (optimal)
  - Difficult to compare directly due to different search spaces
    - Requires different search algorithms
    - Permits different tweaks and optimizations
  - In limited tests:
    - HSPr often faster, typically by a factor of 2-6
    - HSP sometimes faster…
  - **Not true for *all* heuristics**!

Automated Planning

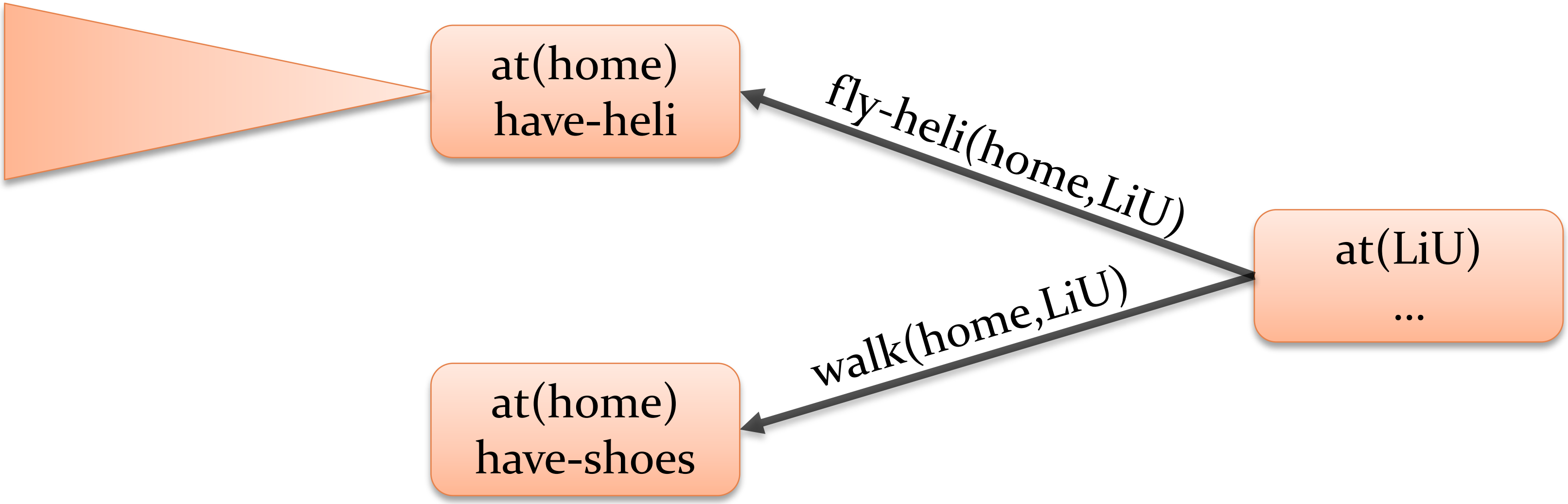
Planning Graphs

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

Recap: Backward Search

BACKWARD SEARCH

- We know if the **effects** of an action can contribute to the goal
- Don't know if we can **reach a state** where its preconditions are true so we can **execute it**



at(home)
have-heli

fly-heli(home,LiU)

at(LiU)
…

walk(home,LiU)

at(home)
have-shoes

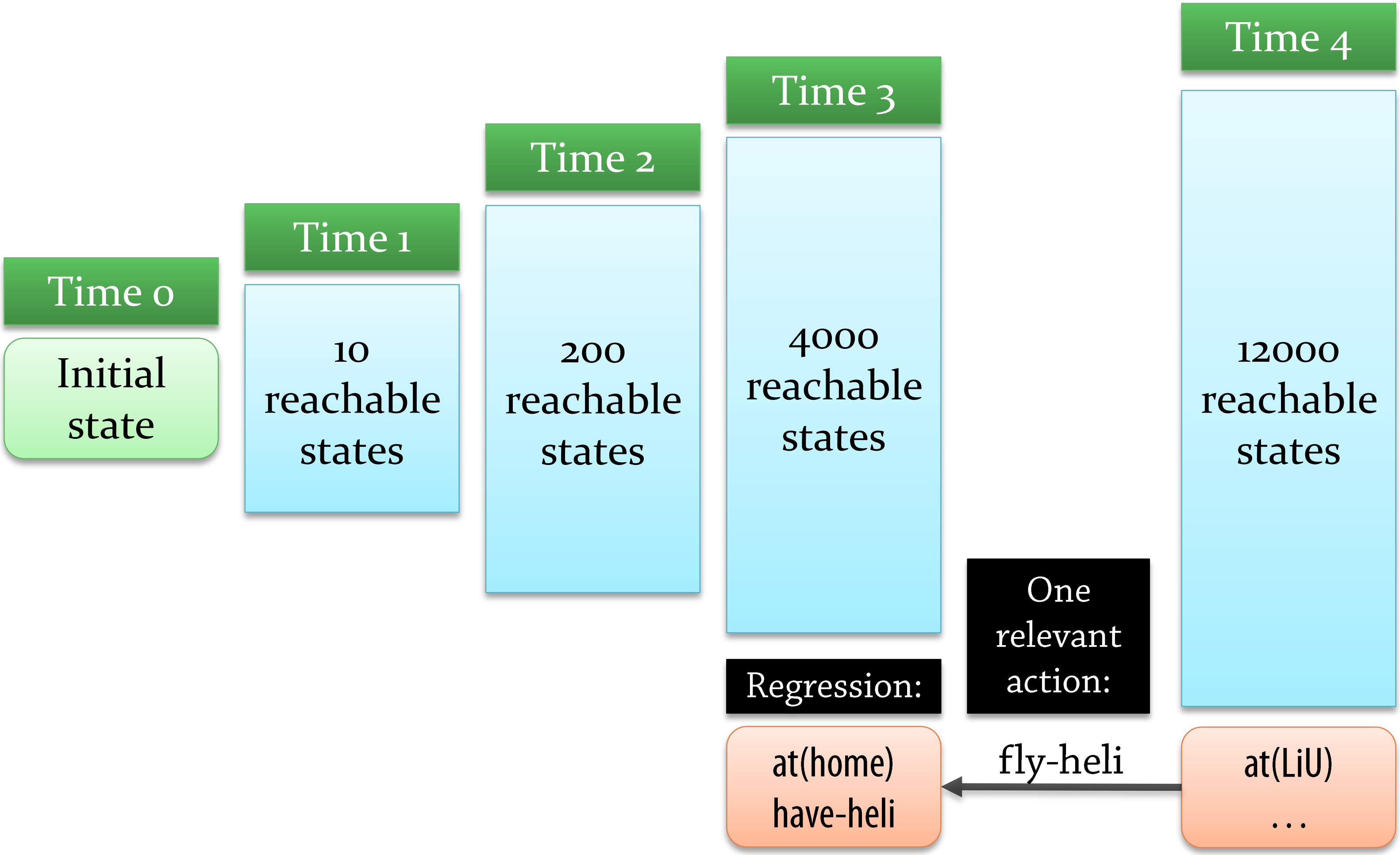
Reachable States

- Suppose we could quickly calculate **all reachable states** at time 0, 1, 2, 3, …

Time 0	Time 1	Time 2	Time 3	Time 4
Initial state	10 reachable states	200 reachable states	4000 reachable states	12000 reachable states
Goal not satisfied	Goal not satisfied in any state	Goal not satisfied in any state	Goal not satisfied in any state	Goal satisfied in 14 of the states: The shortest solution must have 4 actions!

Plan Extraction

- Reachability could be a **<u>pruning filter</u>** in backward search!



Time 0

Initial state

Time 1

10 reachable states

Time 2

200 reachable states

Time 3

4000 reachable states

Time 4

12000 reachable states

Regression:

at(home)
have-heli

One relevant action:

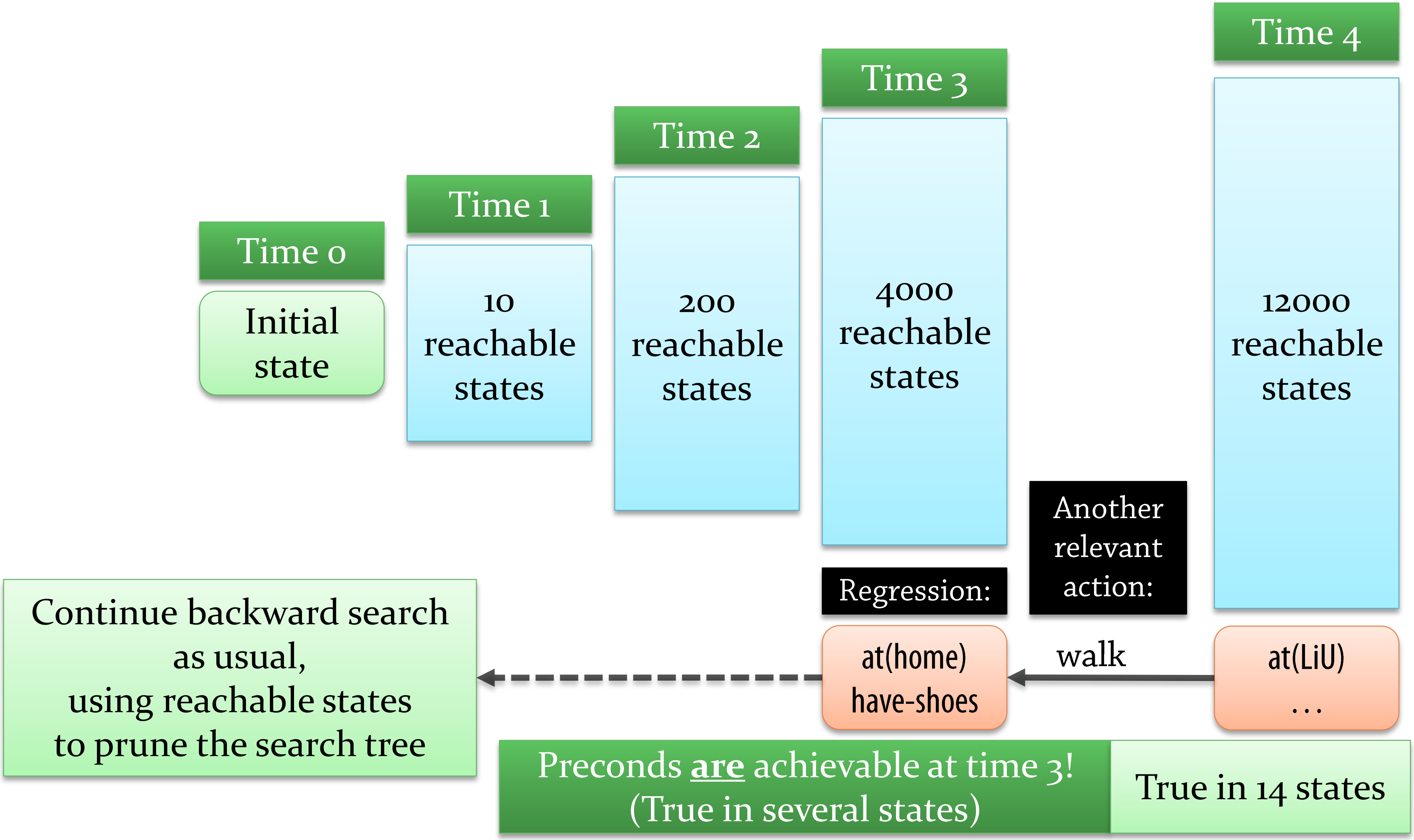
fly-heli

at(LiU)
…

Not true (reachable) in any of the 4000 states at time 3!
We *know* preconds can't be achieved in 3 steps ➔ backtrack

True in 14 states

Plan Extraction (2)



Time 0

Initial state

Time 1

10 reachable states

Time 2

200 reachable states

Time 3

4000 reachable states

Time 4

12000 reachable states

Regression:

Another relevant action:

at(home)
have-shoes

walk

at(LiU)
…

Continue backward search as usual,
using reachable states
to prune the search tree

Preconds **are** achievable at time 3!
(True in several states)

True in 14 states

Problem

Problem: This is not possible!

- ”Suppose we could **quickly** calculate **all reachable states**…”
  - In most cases, calculating **exactly** the reachable states would take far too much time and space…

Possibly Reachable States (1)

- Solution: Don’t be exact!
  - Quickly calculate an **overestimate**

Time 0

Initial state

Time 1

15 *possibly* reachable

10

Time 2

500 *possibly* reachable states

Includes 200

Time 3

15000 *possibly* reachable states

Includes all 4000 truly reachable

Time 4

42000 *possibly* reachable states

Includes all 12000 truly reachable

...out of a billion?

Possibly Reachable States (2)

- Planning algorithm:
  - Keep calculating until we find a timepoint where the goal **might** be achievable

Time 0

Initial state

Goal not satisfied

Time 1

15 *possibly* reachable states

Goal not satisfied in any state

Time 2

500 *possibly* reachable states

Goal not satisfied in any state

Time 3

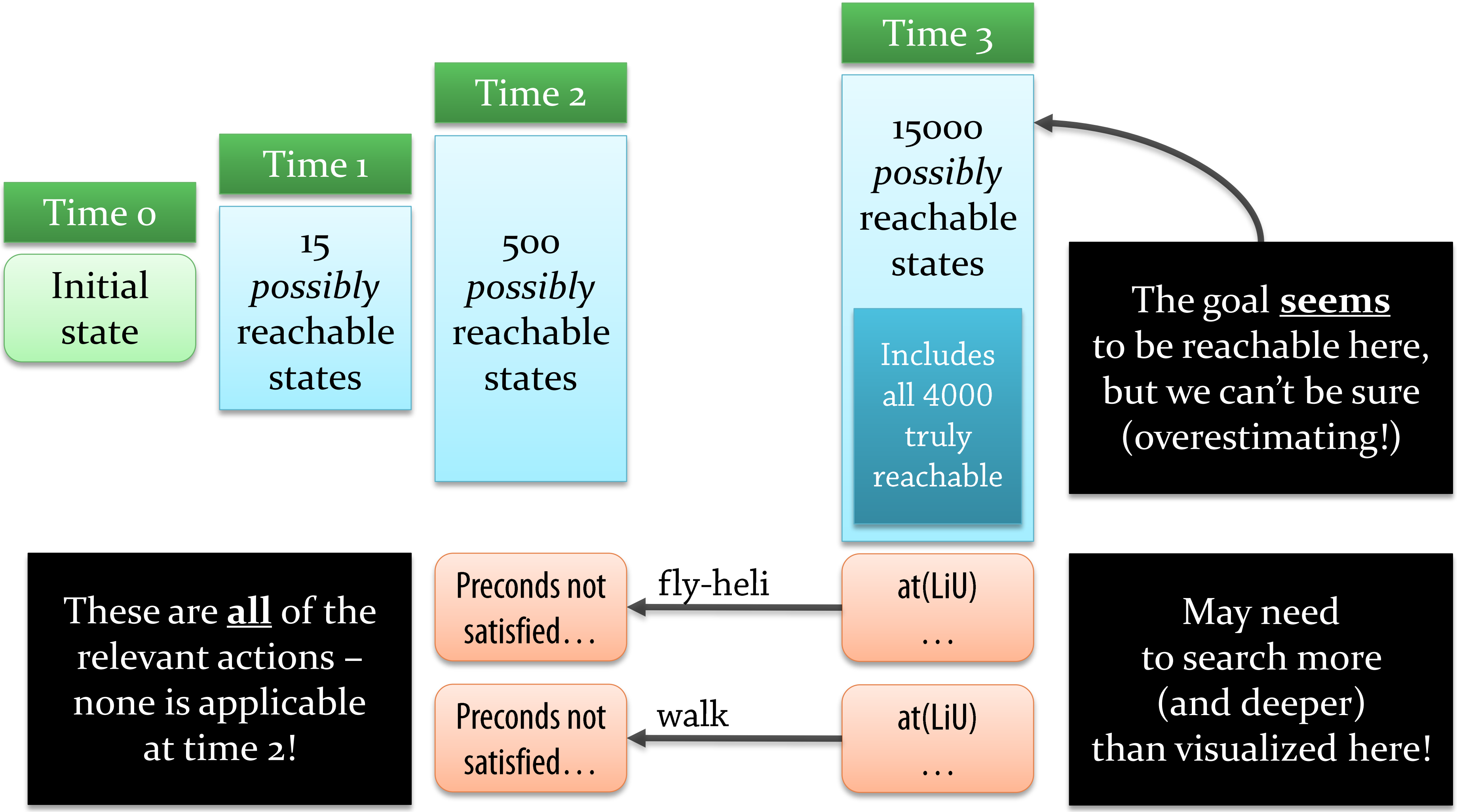
15000 *possibly* reachable states

Includes all 4000 truly reachable

Satisfied in 37 **possibly reachable** states: The shortest solution must have **at least** 3 actions!

Possibly Reachable States (3)

- Backward search will verify what is **truly** reachable
  - In a **much smaller** search space than **plain** backward search



Time 0

Initial state

Time 1

15 *possibly* reachable states

Time 2

500 *possibly* reachable states

Time 3

15000 *possibly* reachable states

Includes all 4000 truly reachable

The goal **seems** to be reachable here, but we can't be sure (overestimating!)

These are **all** of the relevant actions – none is applicable at time 2!

Preconds not satisfied…

fly-heli

at(LiU) …

Preconds not satisfied…

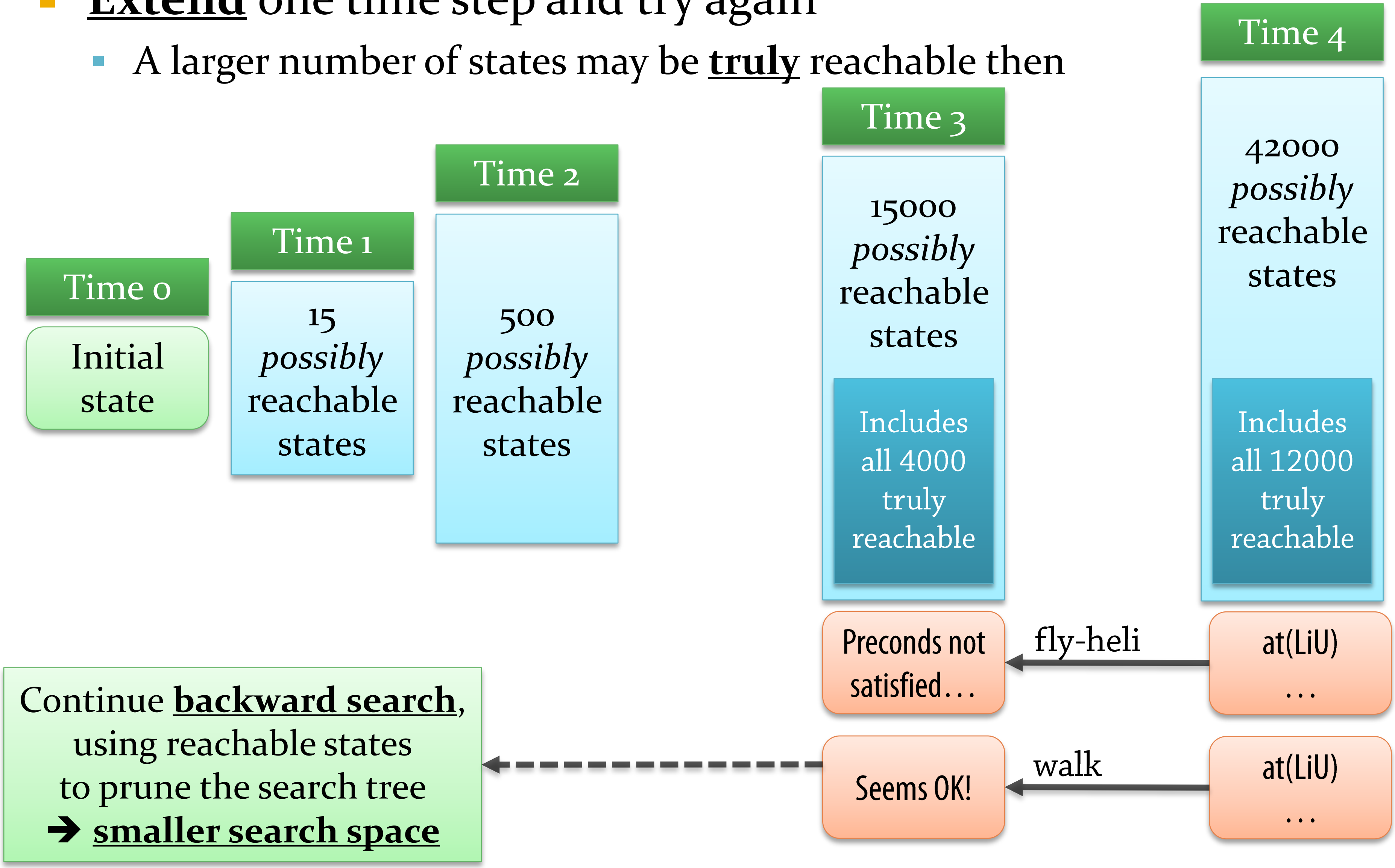
walk

at(LiU) …

May need to search more (and deeper) than visualized here!

Possibly Reachable States (4)

- **Extend** one time step and try again
  - A larger number of states may be **truly** reachable then



Time 0

Initial state

Time 1

15 *possibly* reachable states

Time 2

500 *possibly* reachable states

Time 3

15000 *possibly* reachable states

Includes all 4000 truly reachable

Time 4

42000 *possibly* reachable states

Includes all 12000 truly reachable

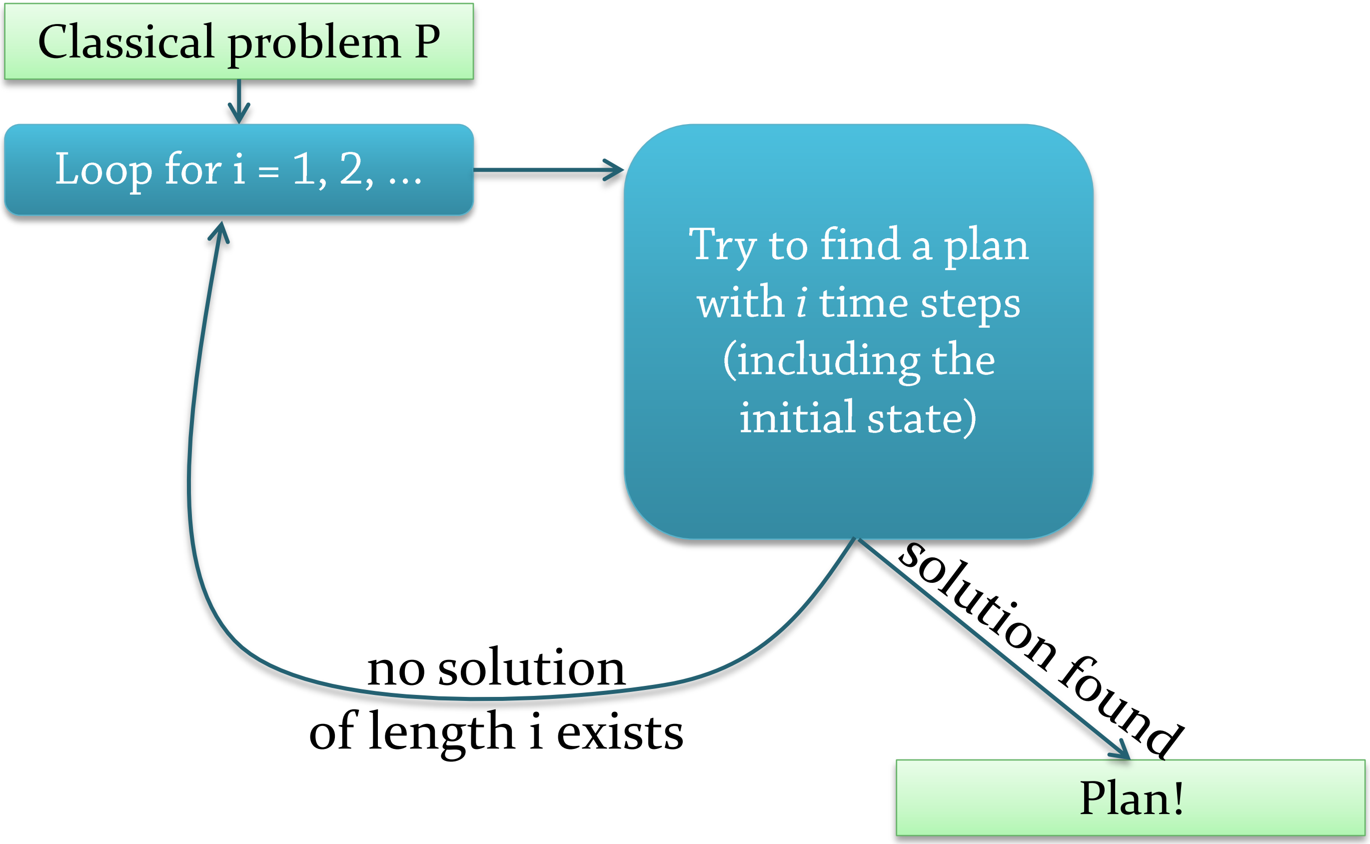
at(LiU) … → fly-heli → Preconds not satisfied…

at(LiU) … → walk → Seems OK!

Continue **backward search**, using reachable states to prune the search tree
➔ **smaller search space**

Iterative Search

- This is a form of **iterative deepening** search!



Classical problem P

Loop for i = 1, 2, …

Try to find a plan with *i* time steps (including the initial state)

no solution of length i exists

solution found

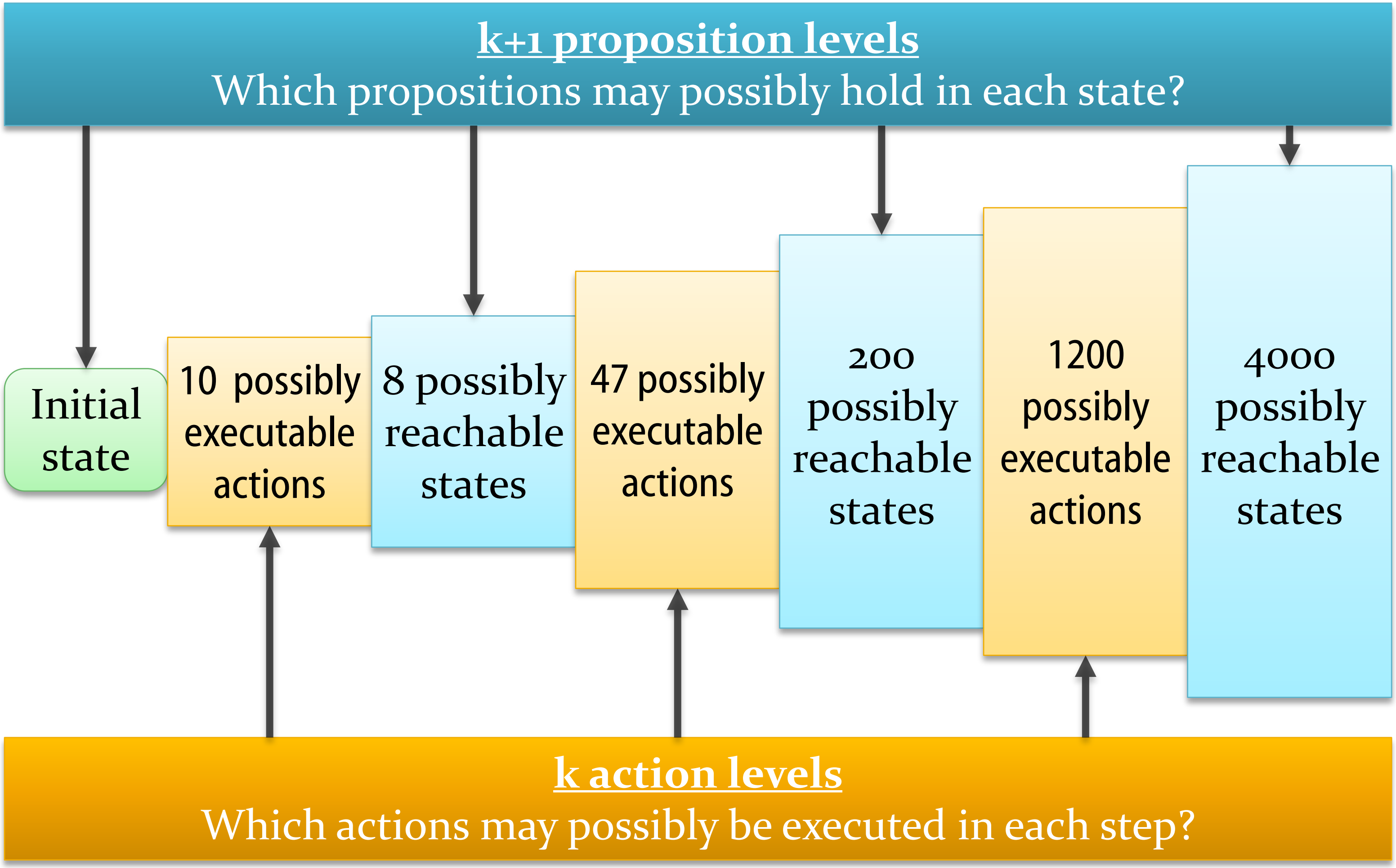
Plan!

$\{$ solutions to P $\} =$
$\{$ solutions to P with i time steps $\mid i \in \mathbb{N}, i > 0 \}$

The GraphPlan Planner

Planning Graph

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



k+1 proposition levels

Which propositions may possibly hold in each state?

Initial state

10 possibly executable actions

8 possibly reachable states

47 possibly executable actions

200 possibly reachable states

1200 possibly executable actions

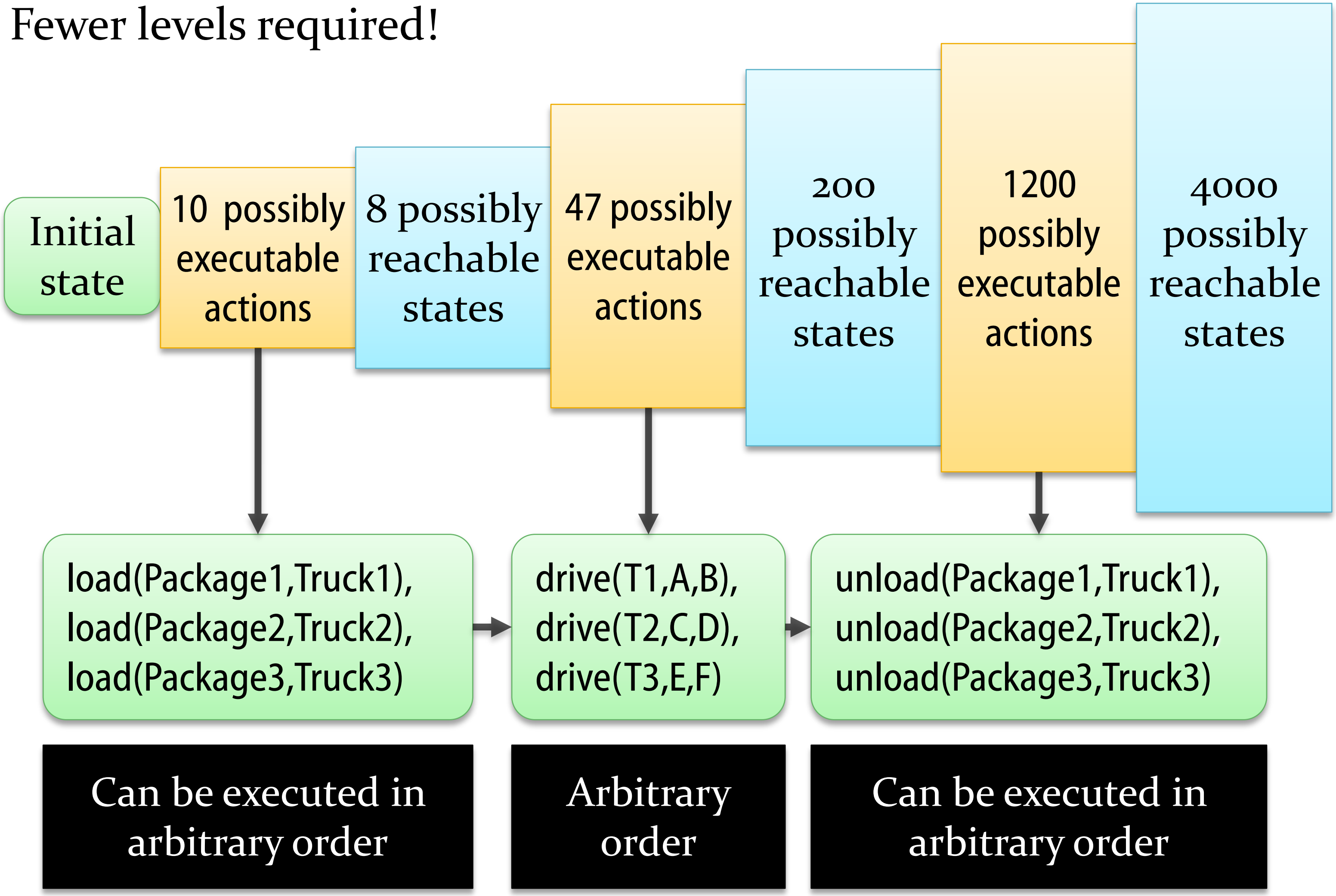
4000 possibly reachable states

k action levels

Which actions may possibly be executed in each step?

GraphPlan: Plan Structure

- **GraphPlan**'s plans are **sequences** of **sets** of actions
  - Fewer levels required!



Initial state

10 possibly executable actions

8 possibly reachable states

47 possibly executable actions

200 possibly reachable states

1200 possibly executable actions

4000 possibly reachable states

load(Package1,Truck1),
load(Package2,Truck2),
load(Package3,Truck3)

drive(T1,A,B),
drive(T2,C,D),
drive(T3,E,F)

unload(Package1,Truck1),
unload(Package2,Truck2),
unload(Package3,Truck3)

Can be executed in arbitrary order

Arbitrary order

Can be executed in arbitrary order

Running Example

- **<u>Running example</u>** 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

- 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}

Can't calculate and store all **reachable states**, one at a time…

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 explicitly consider
sets of actions:
All literals made true by any set
are already there

Not all **combinations** of literals
are reachable,
but every *individual* literal is!

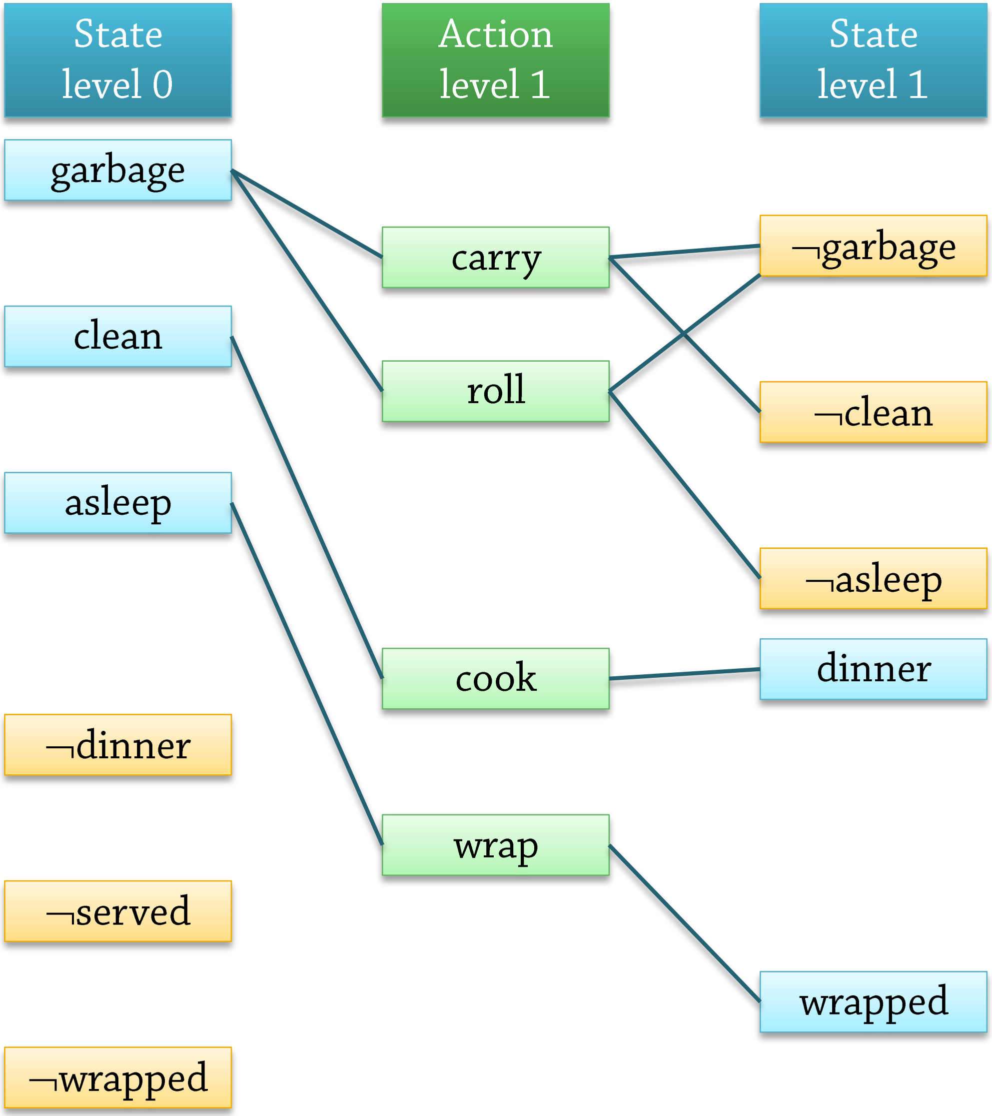
State level 0	Action level 0	State level 1
garbage	Depending on which actions we choose here…	garbage
		¬garbage
clean		clean
		¬clean
asleep		asleep
		¬asleep
		dinner
¬dinner		¬dinner
¬served		¬served
		wrapped
¬wrapped		¬wrapped

Reachable Literals (2)

- Planning Graph Structure:
  - Don't explicitly calculate states
  - For each **applicable action**
    - Add its **effects** to the next state 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…



State level 0

garbage

clean

asleep

¬dinner

¬served

¬wrapped

Action level 1

carry

roll

cook

wrap

State level 1

¬garbage

¬clean

¬asleep

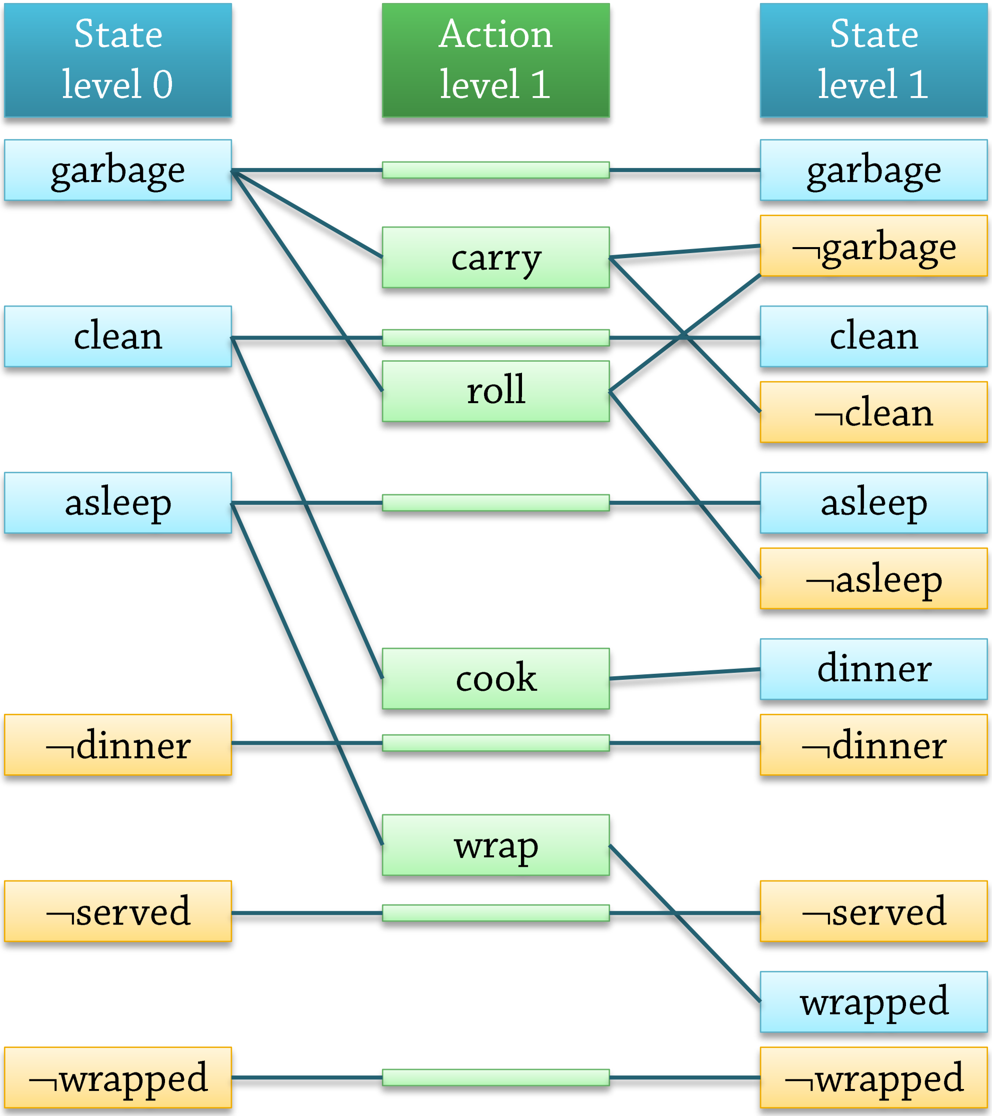
dinner

wrapped

Reachable Literals (3)

- They could **remain** from the previous level!
- To handle this **consistently**, introduce **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
noopdinner	dinner	dinner
noopnotdin	¬dinner	¬dinner
…		



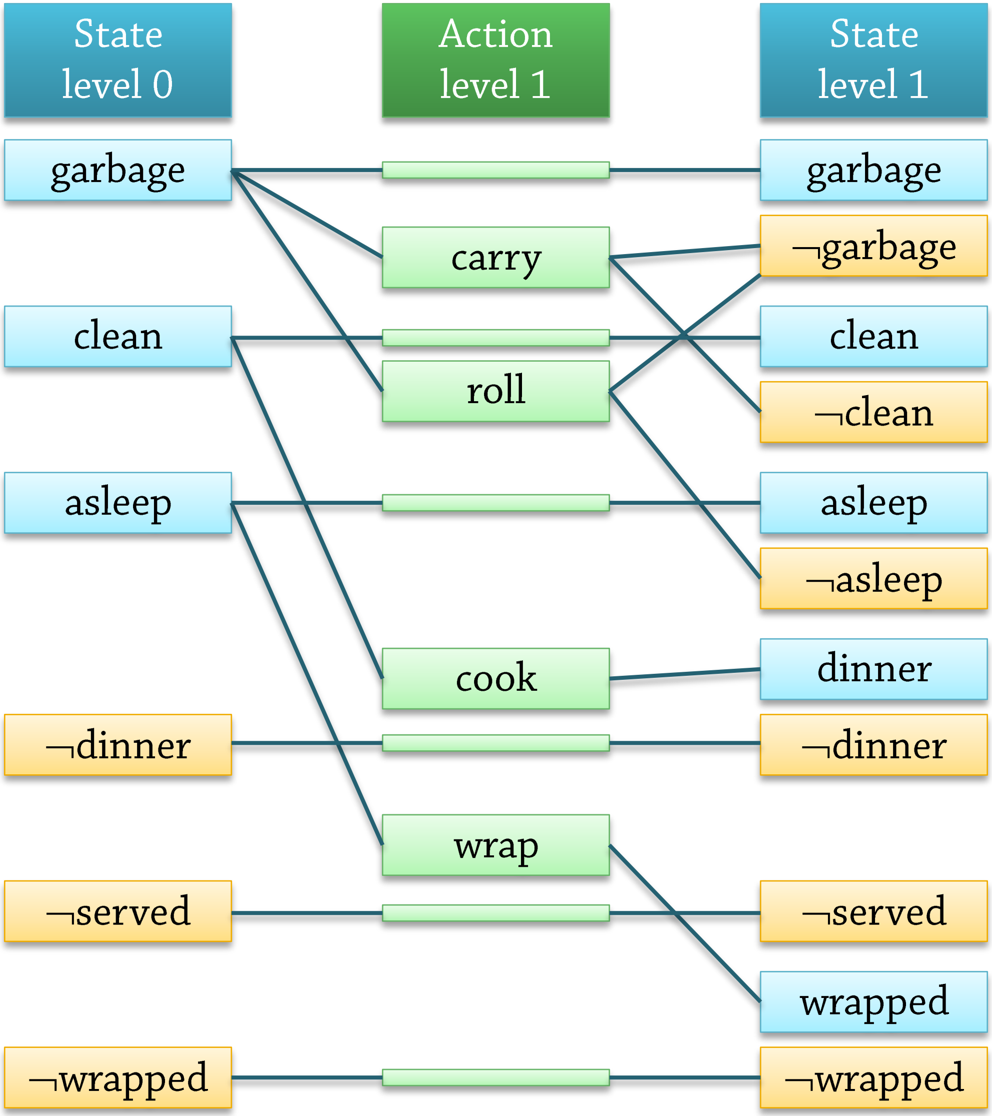
State level 0
garbage
clean
asleep
¬dinner
¬served
¬wrapped

Action level 1
carry
roll
cook
wrap

State level 1
garbage
¬garbage
clean
¬clean
asleep
¬asleep
dinner
¬dinner
¬served
wrapped
¬wrapped

Reachable Literals (4)

- Now the graph is **sound**
  - If an action **might** be executable, it is part of the graph
  - If a literal **might** hold in a given state, it is part of the graph
- But it is quite "**weak**"!
  - Even at state level 1, it seems **any** literal except *served* can be achieved
  - Let's try to add some more information: **Mutual exclusion**



State level 0

garbage

clean

asleep

¬dinner

¬served

¬wrapped

Action level 1

carry

roll

cook

wrap

State level 1

garbage

¬garbage

clean

¬clean

asleep

¬asleep

dinner

¬dinner

¬served

wrapped

¬wrapped

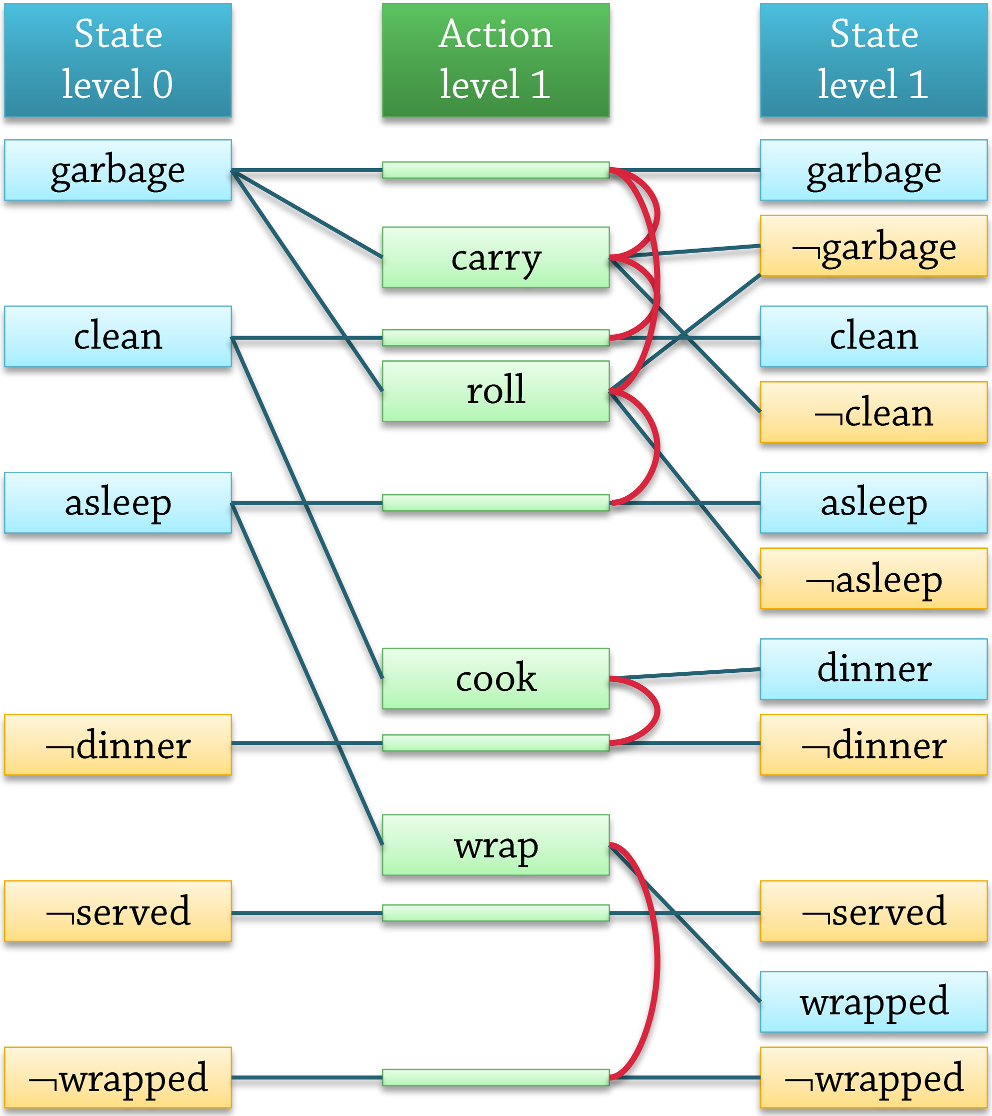
Mutex 1: Inconsistent Effects

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

Two actions in a level are mutex
if their **<u>effects are inconsistent</u>**

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*



State level 0
Action level 1
State level 1

garbage
clean
asleep
¬dinner
¬served
¬wrapped

carry
roll
cook
wrap

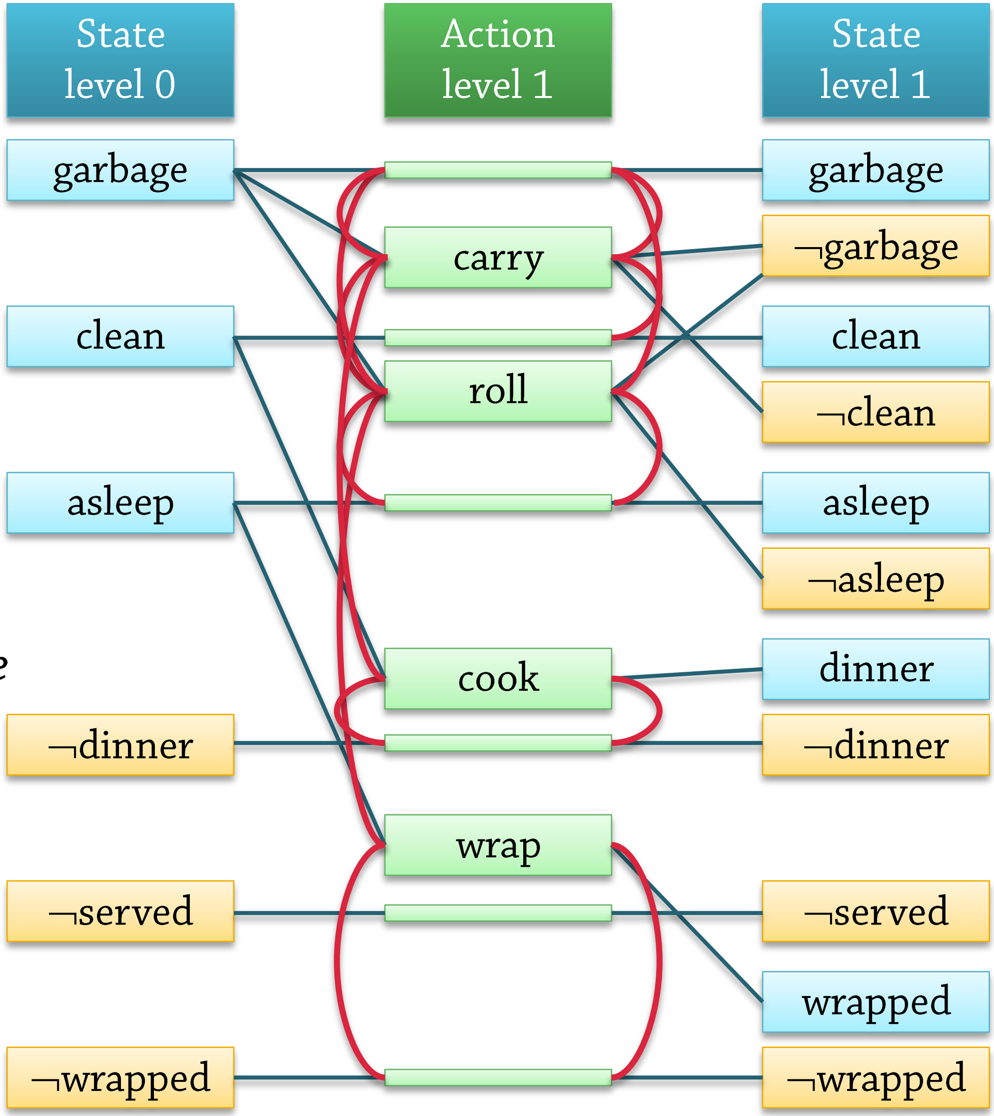
garbage
¬garbage
clean
¬clean
asleep
¬asleep
dinner
¬dinner
¬served
wrapped
¬wrapped

Mutex 2: Interference

Two actions in one level are mutex if **<u>one destroys a precondition</u>** 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*
- …



State level 0

Action level 1

State level 1

garbage

clean

asleep

¬dinner

¬served

¬wrapped

carry

roll

cook

wrap

garbage

¬garbage

clean

¬clean

asleep

¬asleep

dinner

¬dinner

¬served

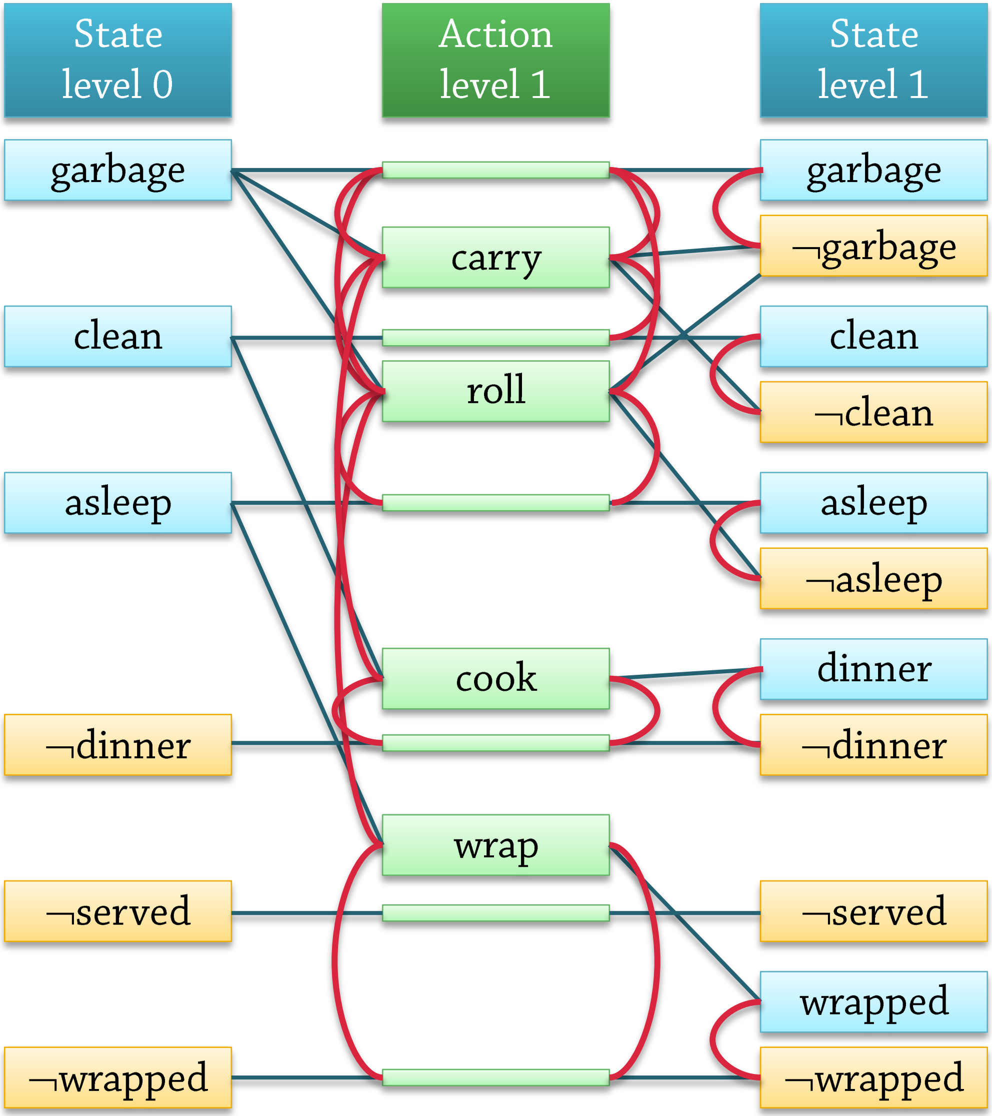
wrapped

¬wrapped

Mutex 3: Inconsistent Support

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

Can't be true at the same time…



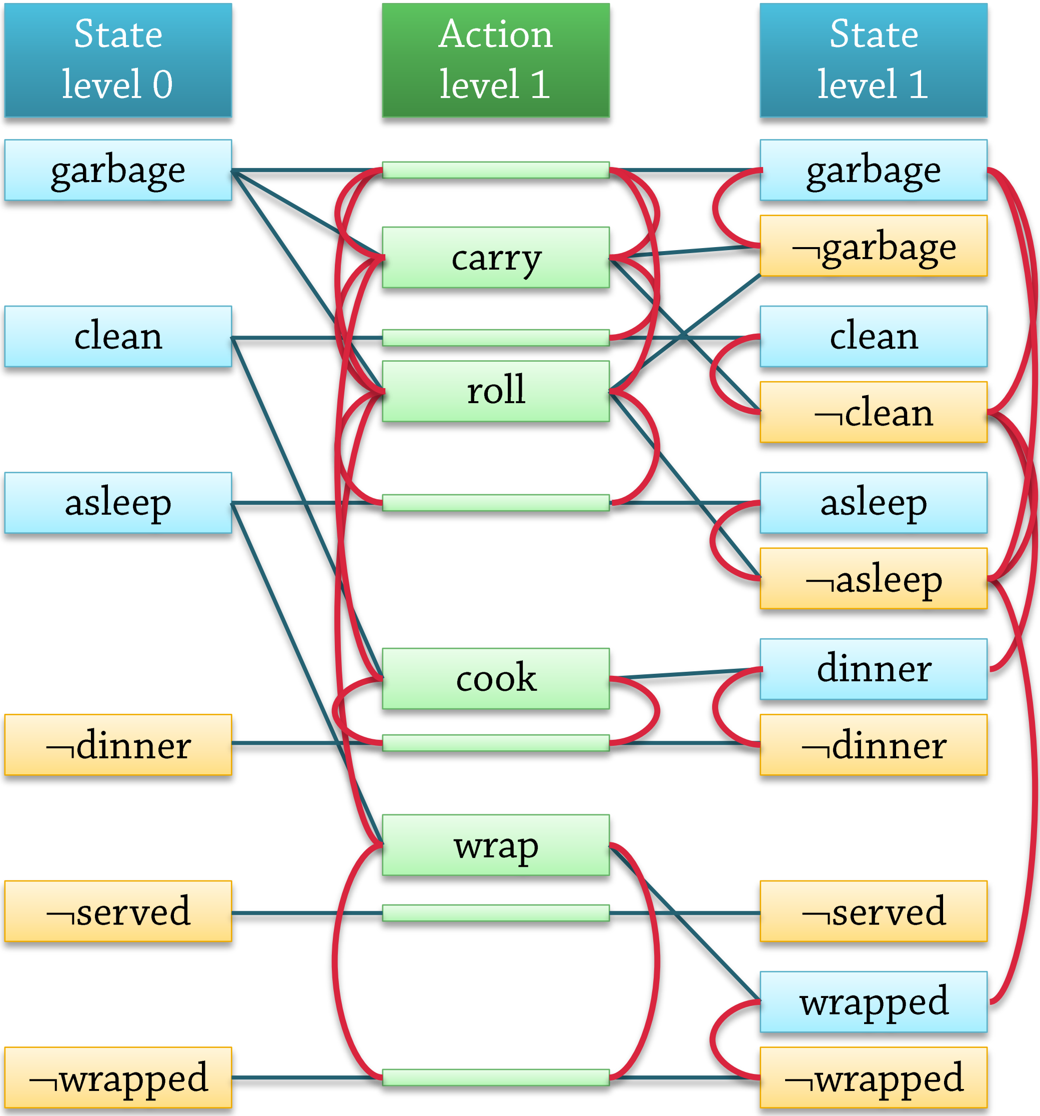
Mutex 4: Inconsistent Support

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

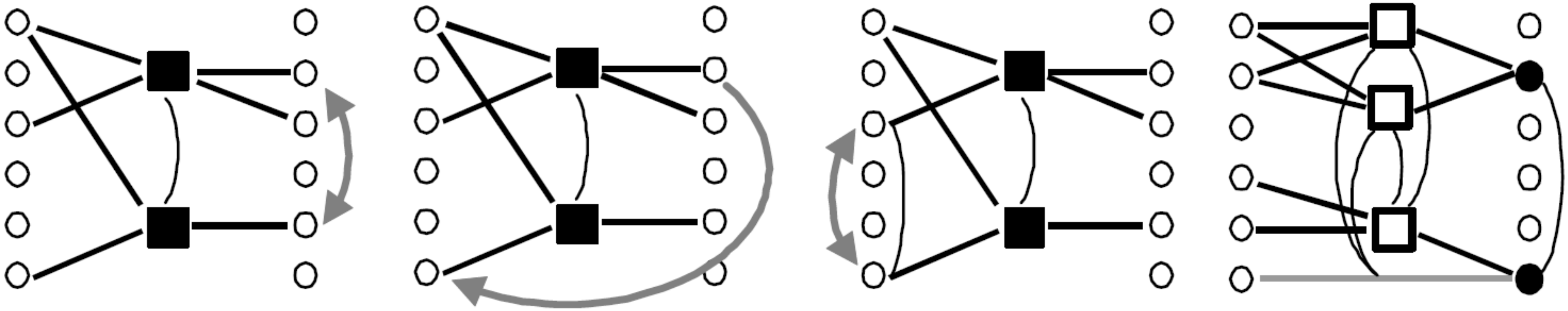


State level 0
garbage
clean
asleep
¬dinner
¬served
¬wrapped

Action level 1
carry
roll
cook
wrap

State level 1
garbage
¬garbage
clean
¬clean
asleep
¬asleep
dinner
¬dinner
¬served
wrapped
¬wrapped

Mutual Exclusion: Summary



Inconsistent Effects

Interference

Competing Needs

Inconsistent Support

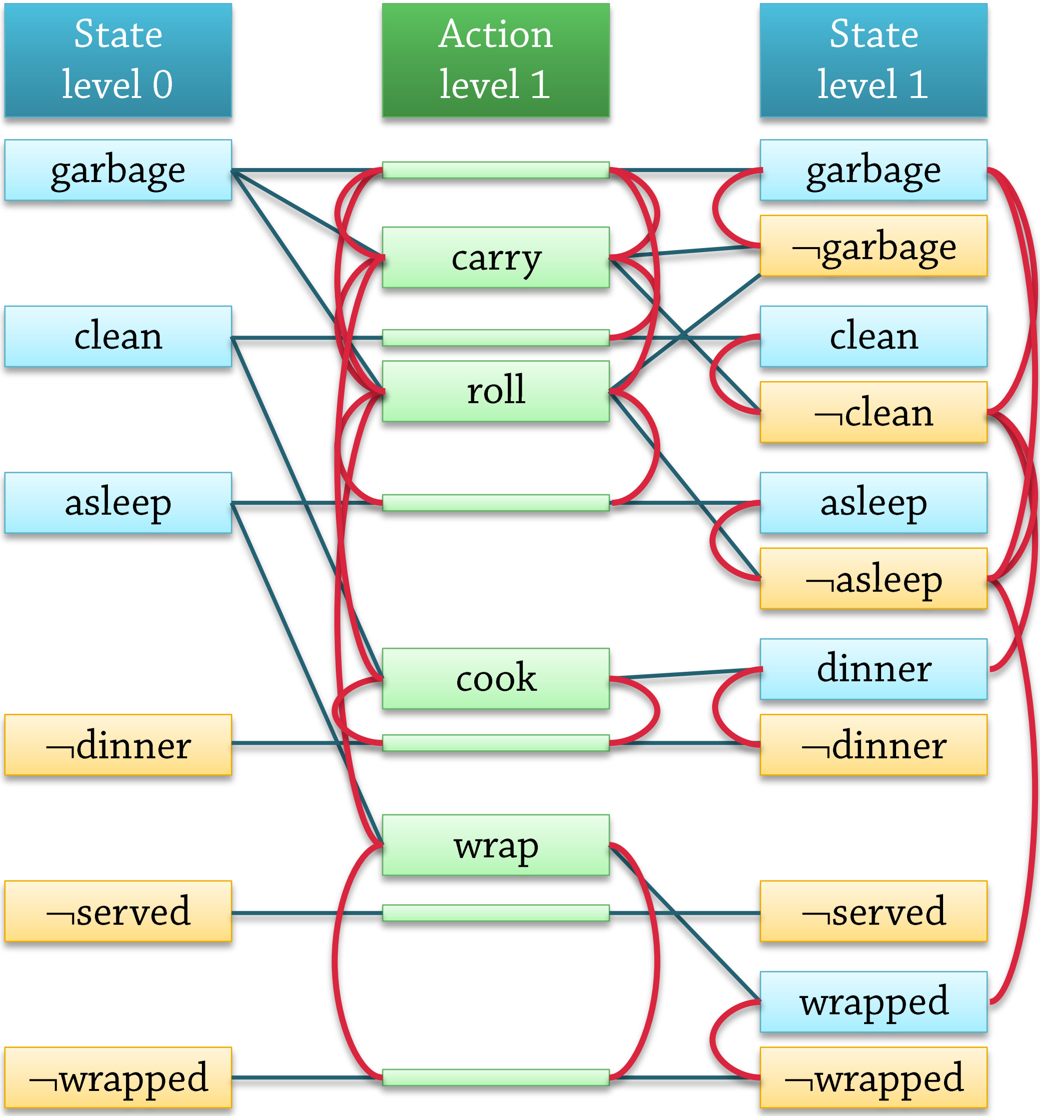
- Two **<u>actions</u>** at the same action-level are mutex if
  - *Inconsistent effects:* an effect of one negates an effect of the other
  - *Interference:* one deletes a precondition of the other
  - ***Competing needs:* they have mutually exclusive preconditions**
- Otherwise they don't interfere with each other
  - Both may appear at the same time step in a solution plan

- Two **<u>literals</u>** 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**

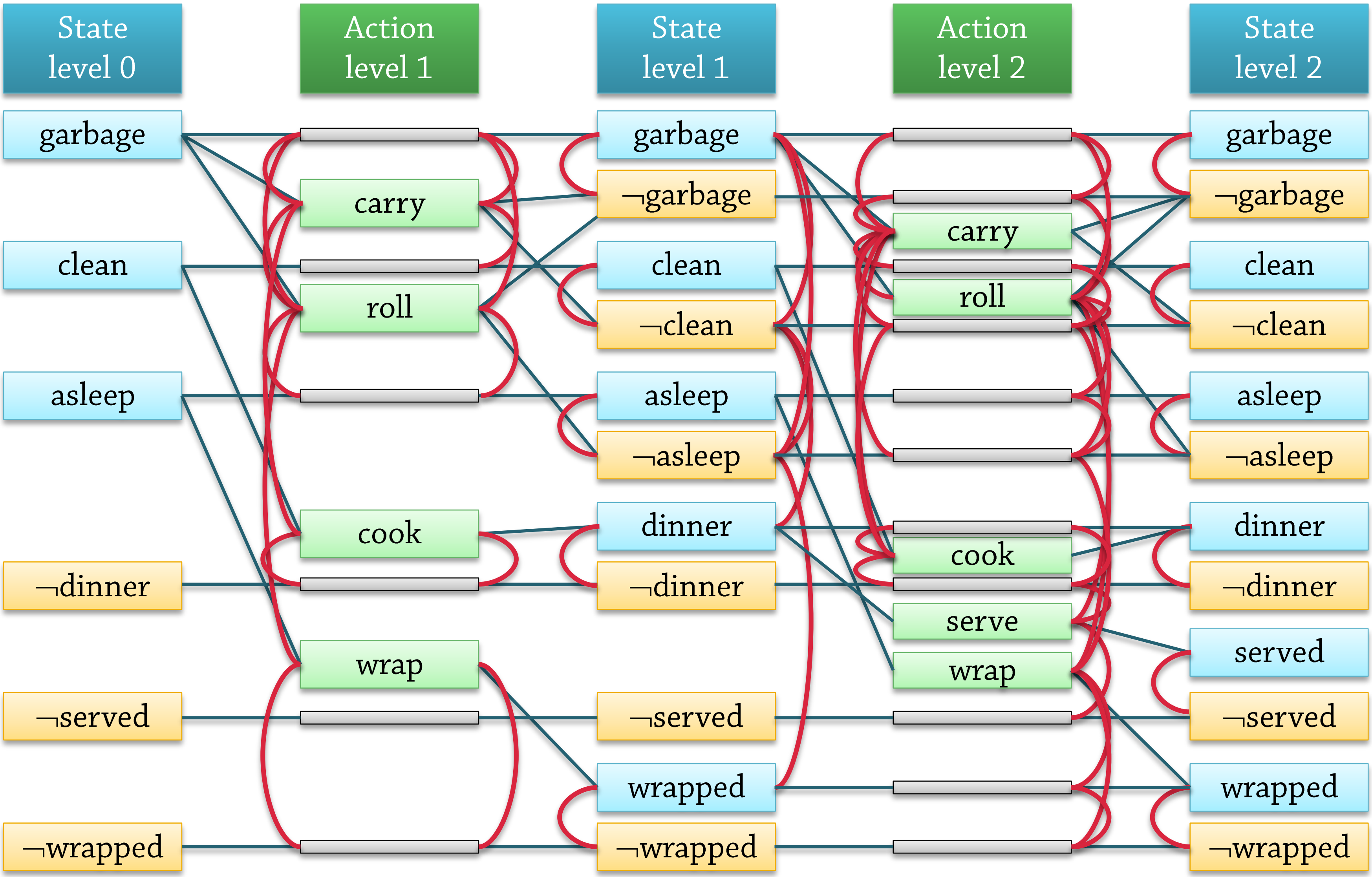
Recursive propagation of mutexes

Early Solution Check

- Is there a possible solution?
  - g = {clean, ¬garbage, served, wrapped}
  - No: We cannot reach a state where served is true in a single (parallel) step

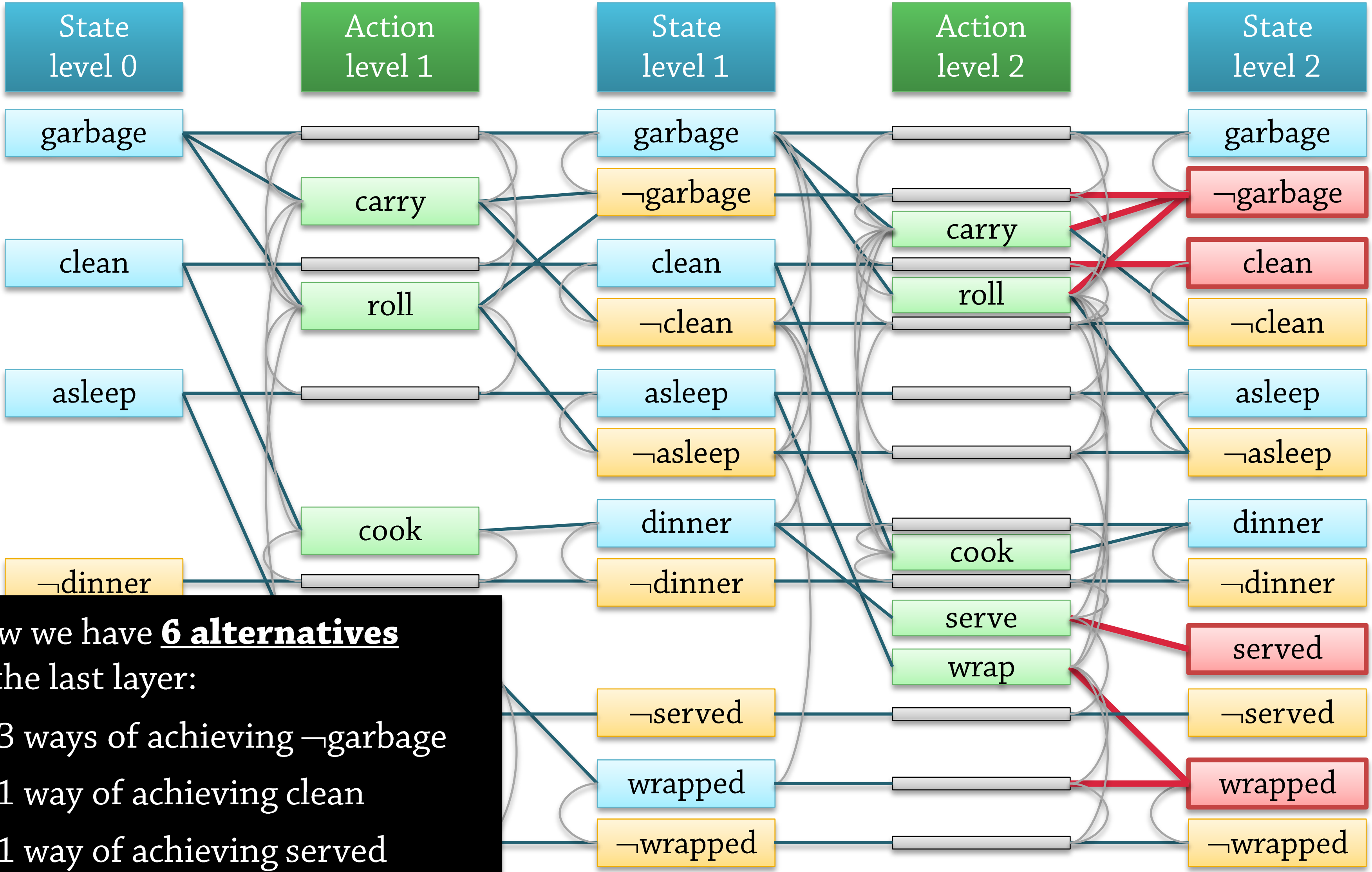


Expanded Planning Graph



All goal literals are present in level 2, and none of them are mutex!

Solution Extraction (1)

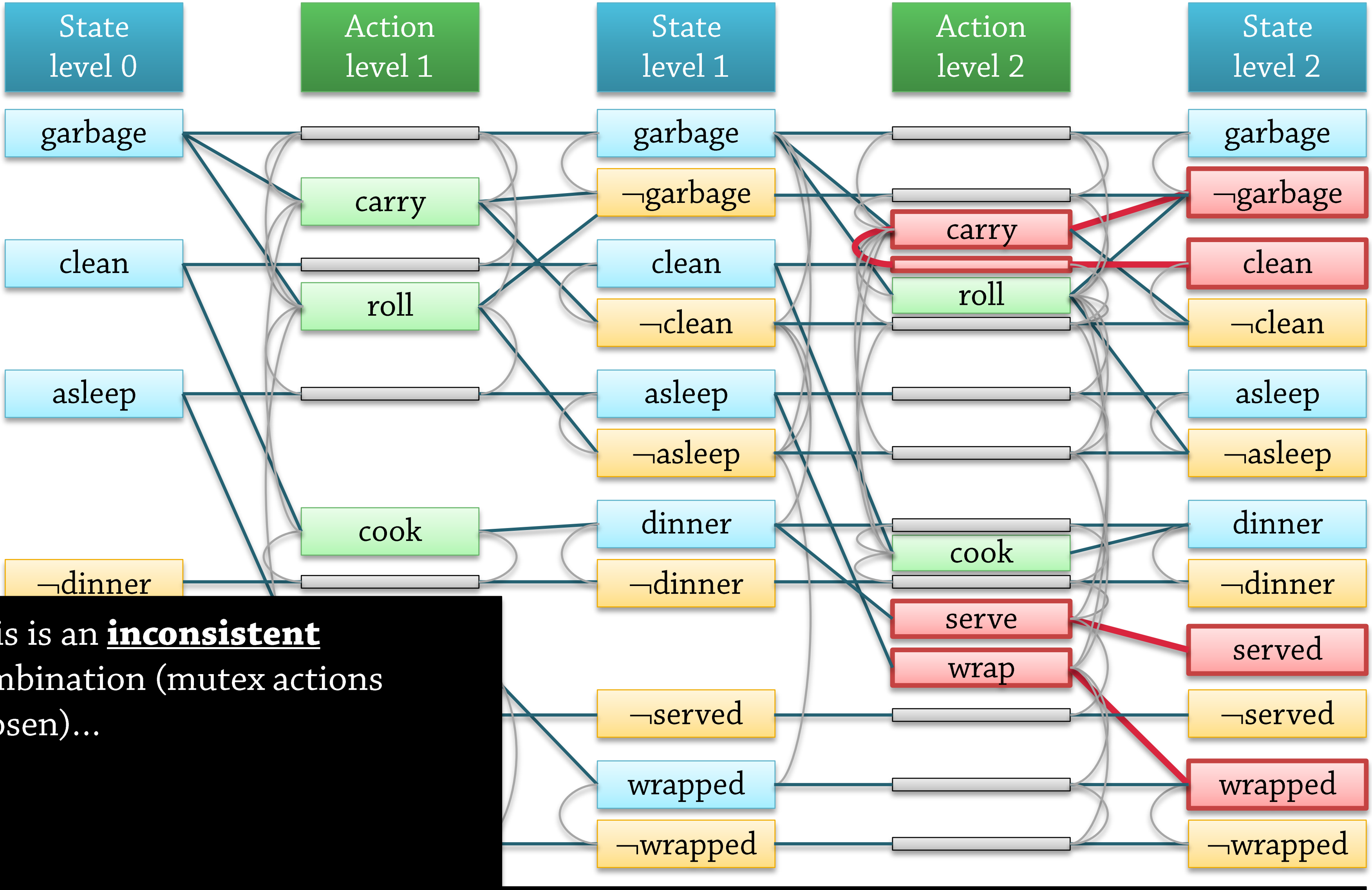


Now we have **6 alternatives** at the last layer:

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

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

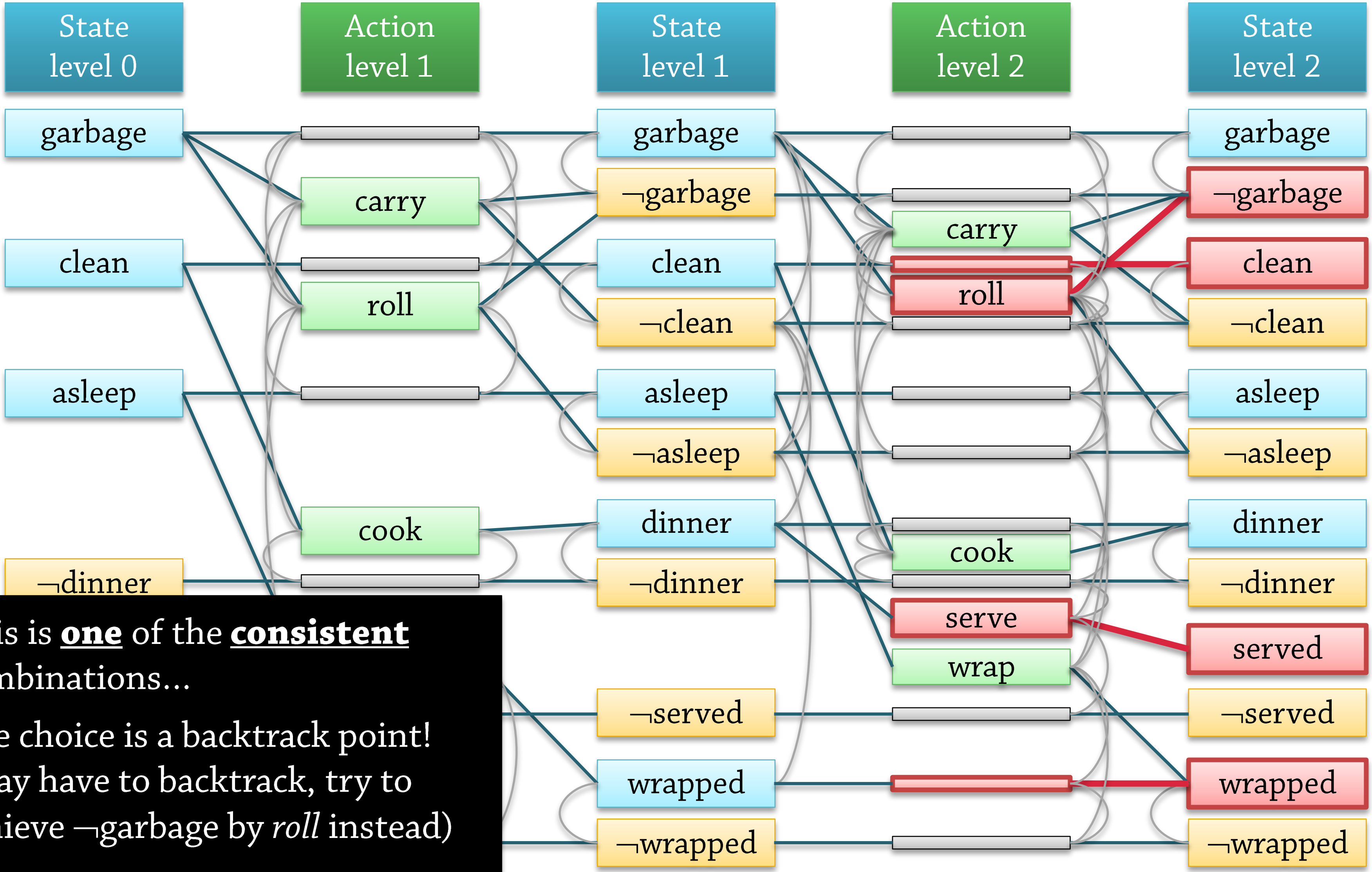
Solution Extraction (2)



This is an ***inconsistent*** combination (mutex actions chosen)…

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

Solution Extraction (3)

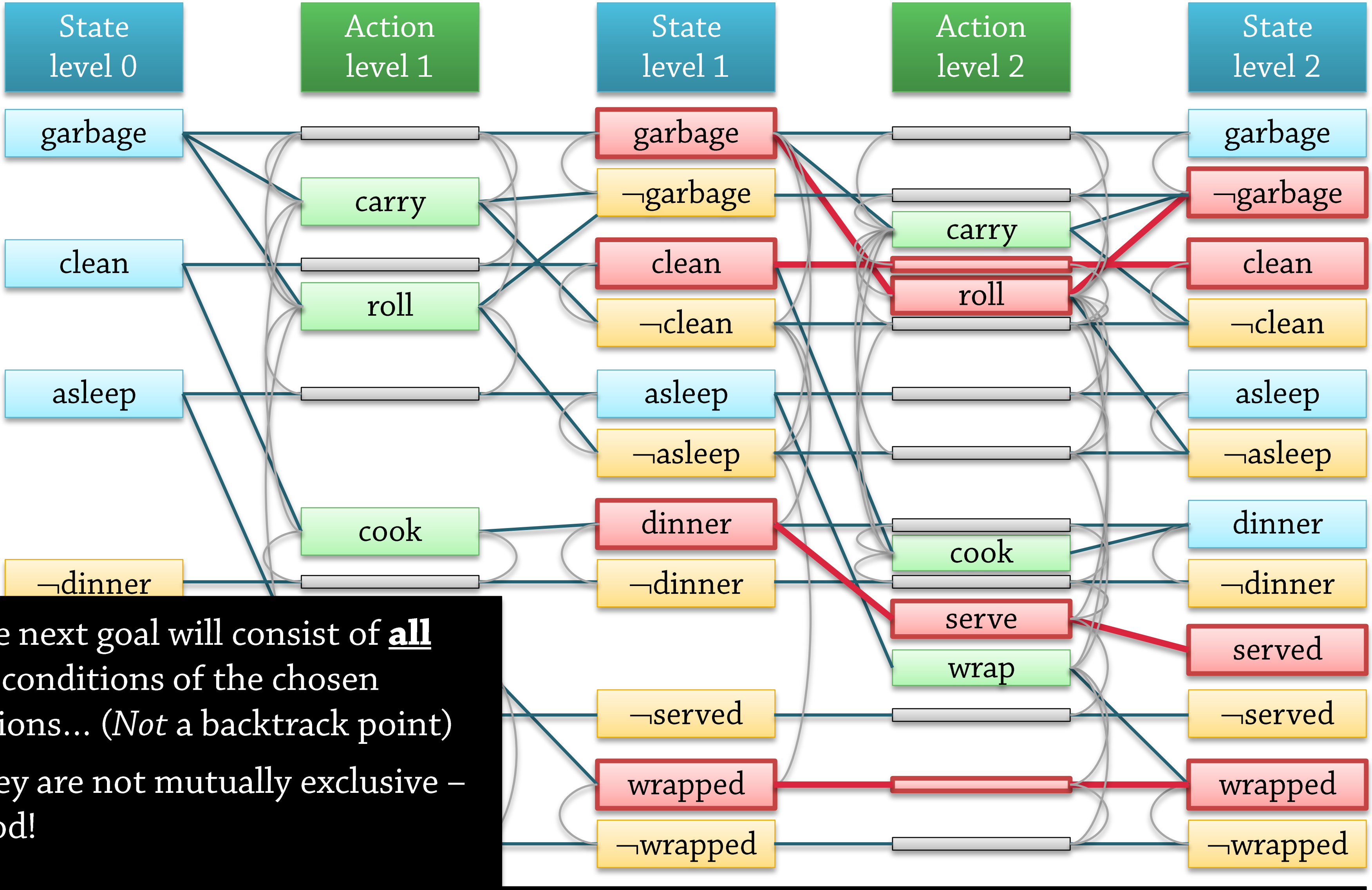


This is **one** of the **consistent** combinations…

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)

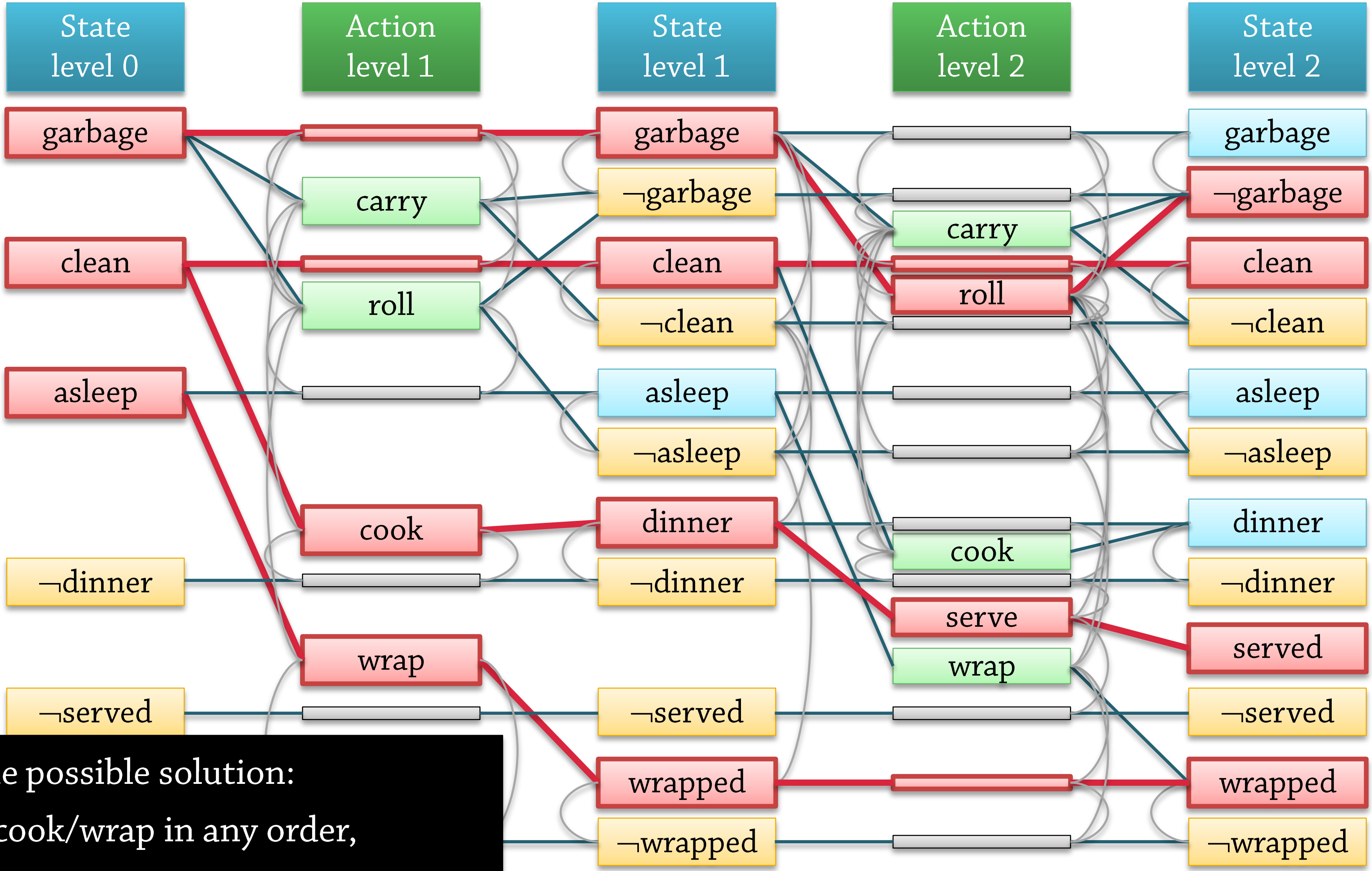


The next goal will consist of **all** preconditions of the chosen actions… (*Not* a backtrack point)

They are not mutually exclusive – good!

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

Solution Extraction (5)



One possible solution:

1) cook/wrap in any order,

2) serve/roll in any order

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

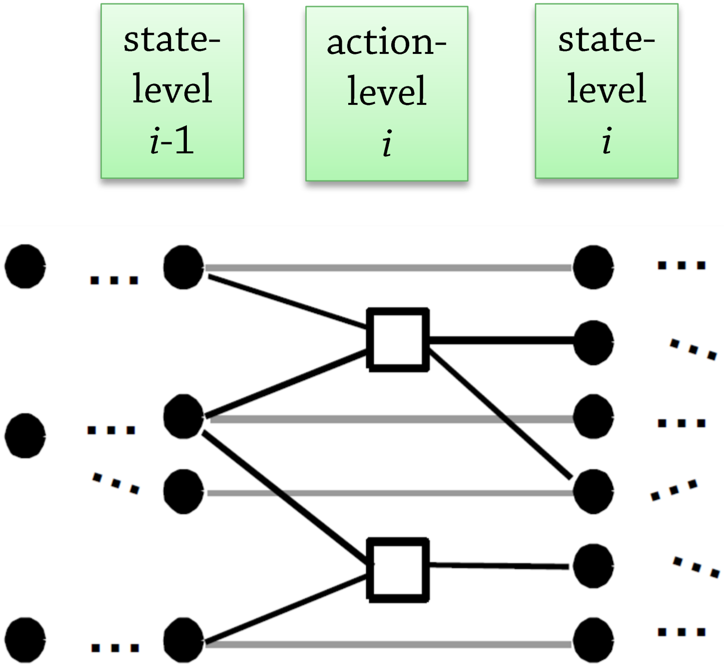
Solution Extraction 6

The set of goals we are trying to achieve

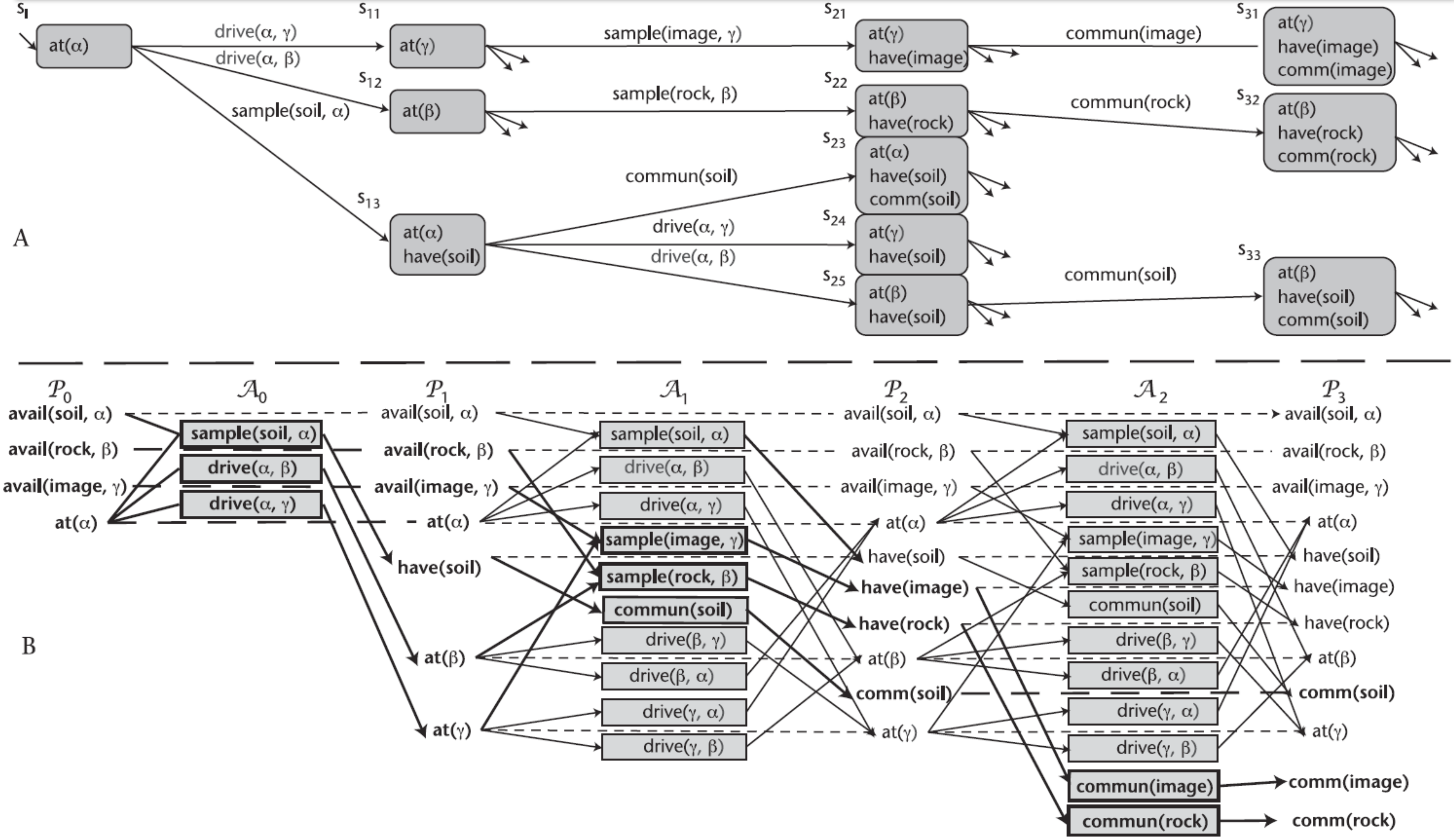
The level of the state s_i, starting at the highest level

```
procedure **Solution-extraction**($g$,$i$)
    **if** $i$=0 then return the solution
    **nondeterministically choose**
        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!)
    **if** 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 **backwards search**, but only among the actions in the graph (generally much fewer)!



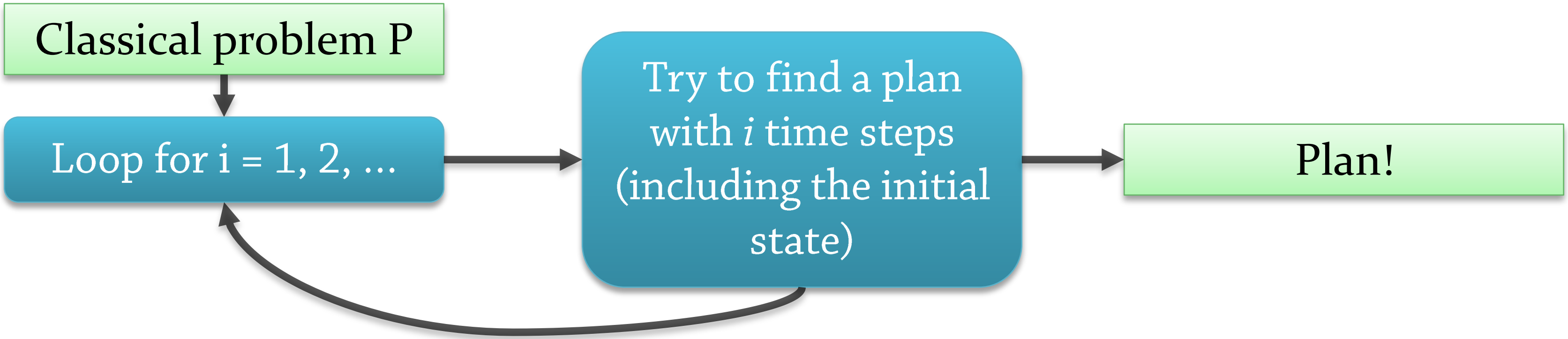
Example Planning Graph for Rover problem (mutexes not shown)



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

Parallel Optimality

- A form of iterative deepening:



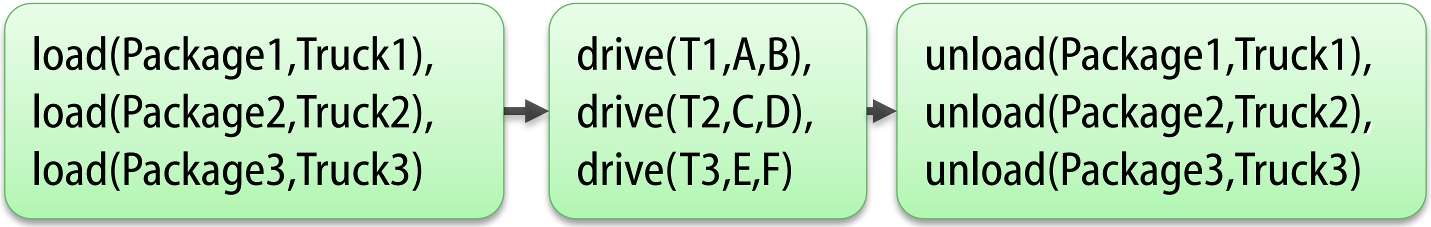
Classical problem P

Loop for i = 1, 2, …

Try to find a plan with *i* time steps (including the initial state)

Plan!

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



load(Package1,Truck1),
load(Package2,Truck2),
load(Package3,Truck3)

drive(T1,A,B),
drive(T2,C,D),
drive(T3,E,F)

unload(Package1,Truck1),
unload(Package2,Truck2),
unload(Package3,Truck3)

Relaxed Planning Graph Heuristics in FastForward (FF)

Introduction

Heuristics as approximations of h+ (optimal DR)

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

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

Optimistic:
As if achieving the *most expensive* goal
would *always* achieve the others

Gives far too little information

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

cost(p and q) = cost(p) + cost(q)

Pessimistic:
As if achieving one goal
could *never* help in achieving the others

Informative,
but can exceed even h* by a large margin!

How can we take some positive interactions into account?

FF Heuristics (1)

- The **planning graph** takes positive interactions into account
  - Creating a full planning graph for every visited state? Too slow, but…

Let's apply **delete relaxation** to the planning graph!

(Technique pioneered by FastForward, FF)

  - No delete effects ➔ **no mutexes** to calculate
    (no inconsistent effects, no interference, …)
  - No mutexes exist ➔ **fewer levels** required
  - No mutexes exist ➔ **no backtracking** needed in solution extraction
  - Can extract a **Graphplan-optimal** relaxed plan in **polynomial** time

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

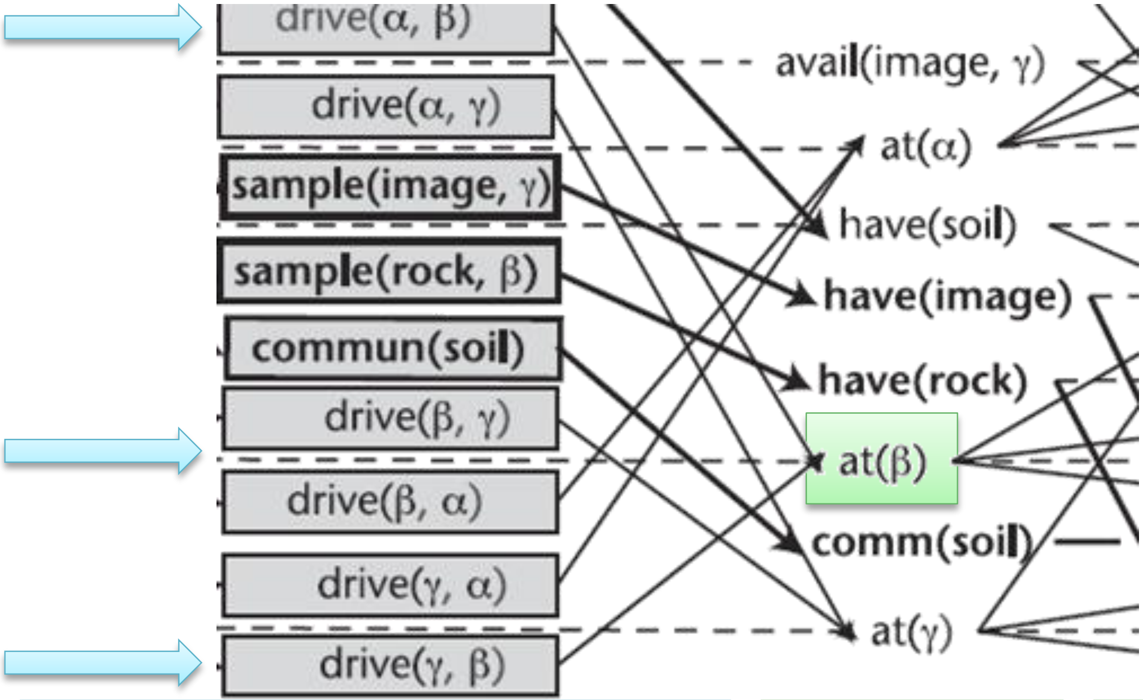
FF Heuristics (2)

But calculating h+ 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*
- Still, the delete-relaxed plan *can* take positive interactions into account
  - ➔ Often closer to true costs than h_{add}
- Plan extraction can use several heuristics (!)
  - Trying to reduce the *sequential* length of the relaxed plan

Plan Extraction Heuristics (1)

- Recall that plan extraction uses **backward search**
  - For each goal fact at a given level, we must find an achiever

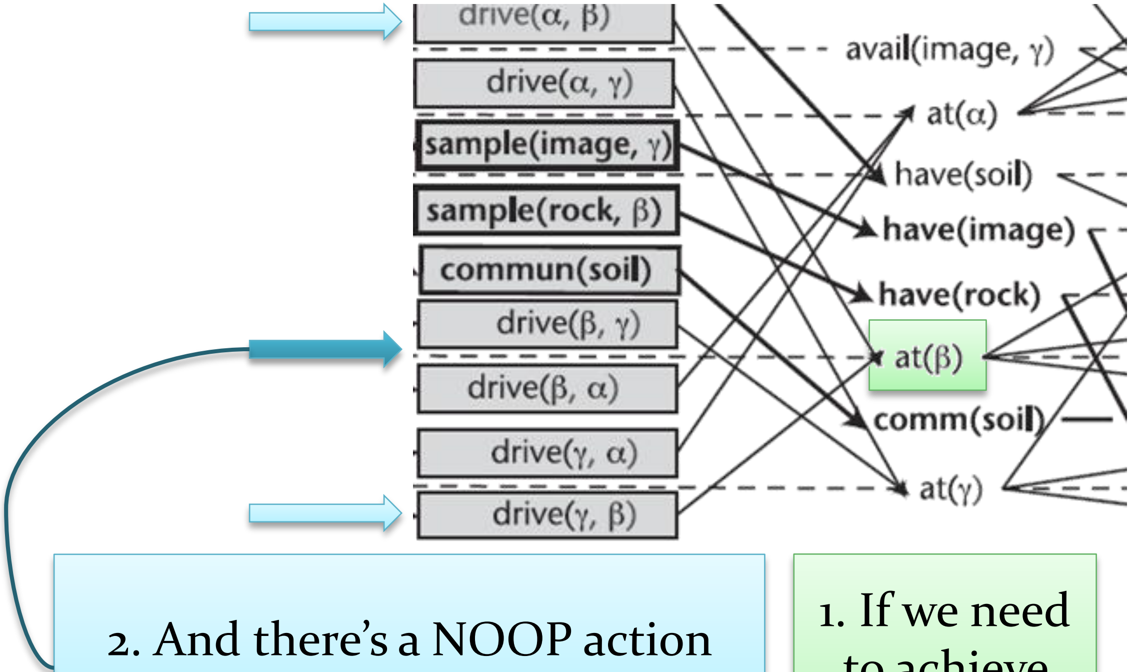


3. Here, at least one of drive(α,β), noop-at-β, drive(γ,β) must be the achiever

2. Here, at(β) must be achieved

1. Here, some action has at(β) as a precondition

Plan Extraction Heuristics (1): Noop



1. If we need to achieve this fact…

2. And there's a NOOP action available in the preceding level…

- Then use the noop action as the achiever
  - Achieve every fact as early as possible!
- In a *delete-relaxed problem*, this is always possible
  - There is a *noop action* ➔ The fact *can* be achieved earlier
  - There are no delete effects ➔ No action can *conflict* with achieving it earlier

Plan Extraction Heuristics (2)

- **Difficulty Heuristic**:
  - If there is no maintenance action available for *f* in level i-1, we should choose an action that seems "easy"
  - Intuition:
    - The difficulty of achieving *one* precondition *p* of an action *a* corresponds to the *first layer* at which *p* can first be achieved
  - Define:
    - difficulty(a) = sum { index of first fact layer where p appears | p is a precondition of a }
  - Select an action with minimal difficulty

Simplified Plan Extraction

```
function ExtractPlan(plan graph $P_0A_0P_1…A_{n-1}P_n$, goal G)
    for i = n . . . 1 do
        $G_i$ ← goals reached at level i
    end for
    for i = n . . . 1 do
        for all $g \in G_i$ not marked ACHIEVED at time i do
            find min-difficult $a \in A_{i-1}$ s.t. $g \in add(A_{i-1})$
            $RP_{i-1}$ ← $RP_{i-1} \cup \{a\}$ // Add to the relaxed plan
            for all $f \in prec(a)$ do
                $G_{layerof(f)} = G_{layerof(f)} \cup \{ f \}$
            end for
            for all $f \in add(a)$ do
                mark f as ACHIEVED at times i – 1 and i
            end for
        end for
    end for
    return RP // The relaxed plan
end function
```

Partition the goals of the problem instance depending on the level where they are first reached

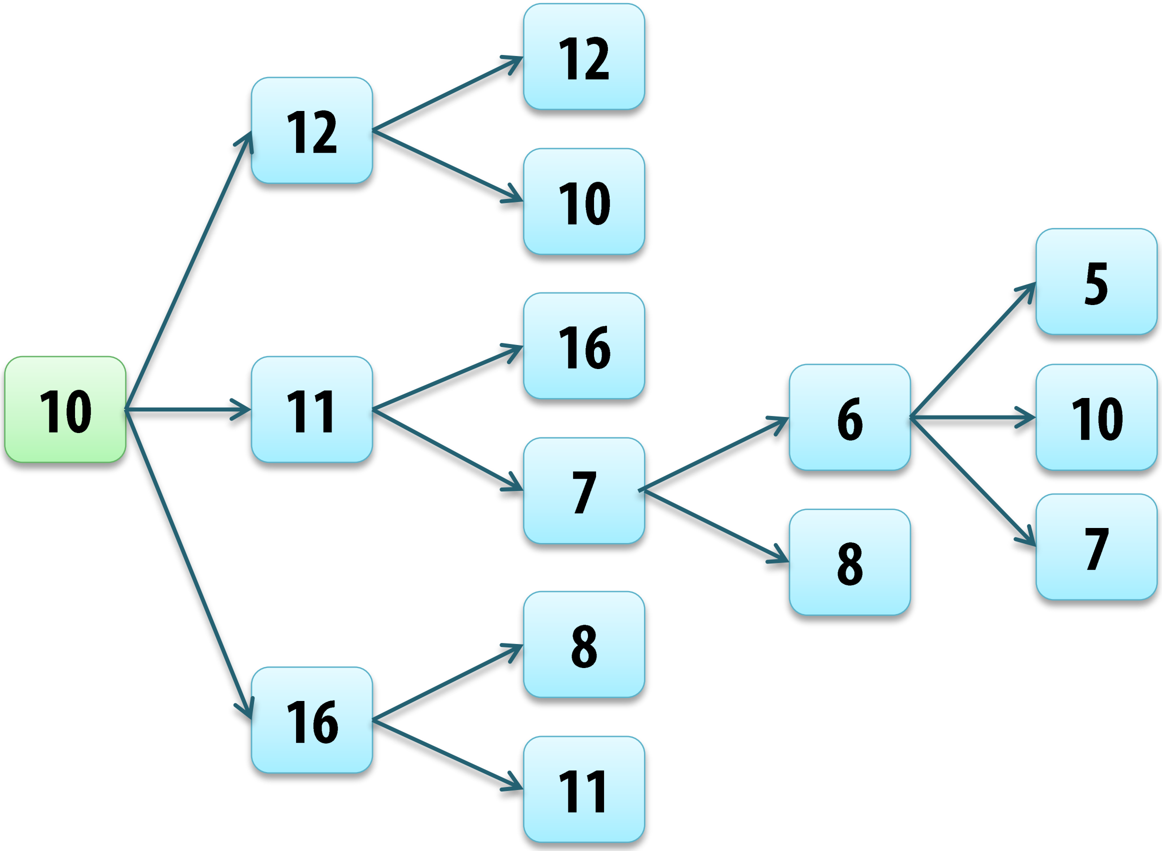
All goals that could not be reached before level i must be reached at level i

Must achieve prec(a) at some time! Heuristic: Do this at the first level we can – layerof(f).

One action can achieve more than one goal – mark them all as "done"

FF: Enforced Hill-Climbing

- FF also pioneered **enforced hill-climbing** (EHC)
  - Breadth-first search until you find a node with better heuristic value



Compared to standard HC:

More persistent.

More systematic when searching for better nodes.

None better than 10:
Expand **all**!

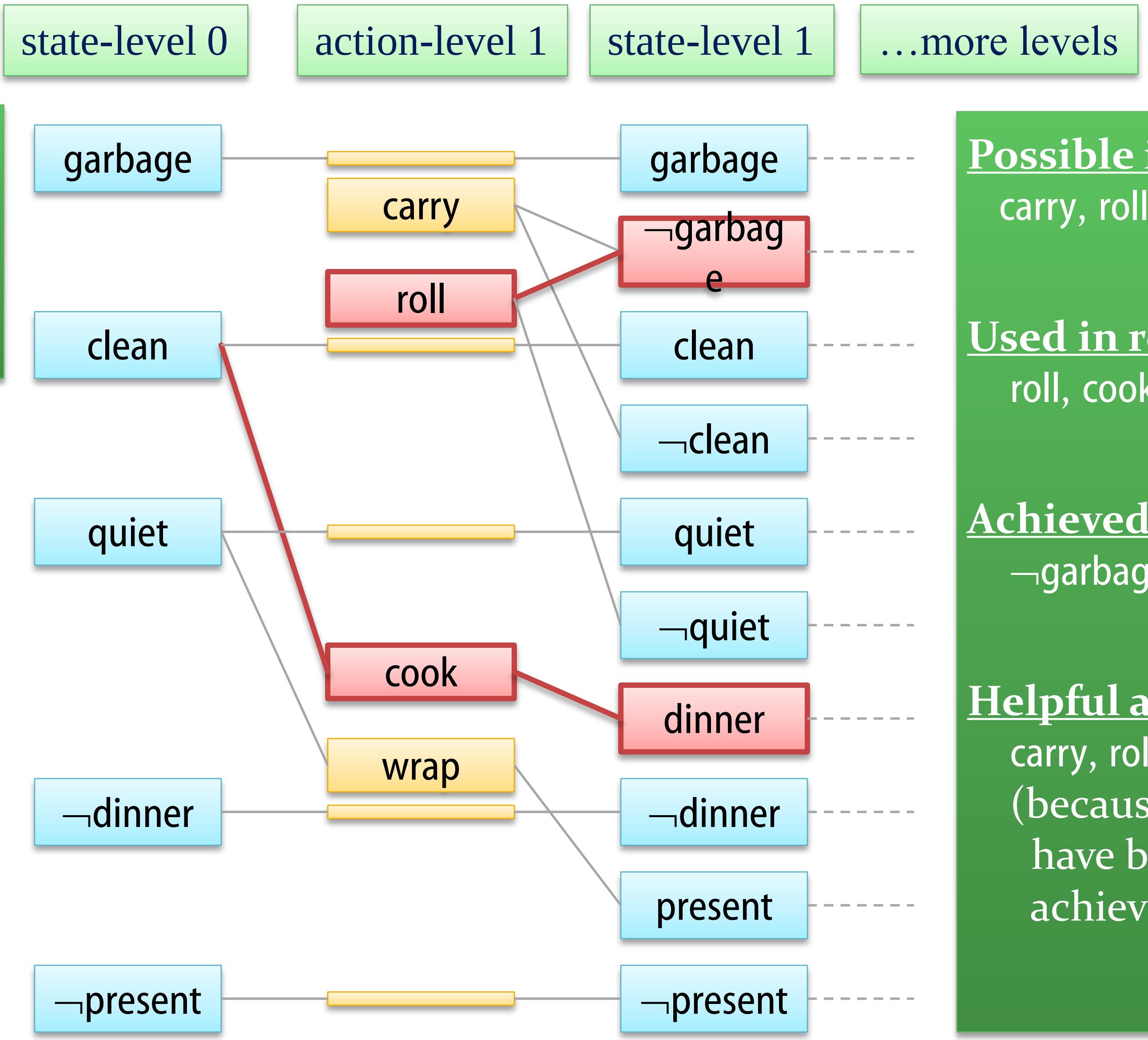
Some better than 10:
Choose the **best**

FF: Helpful Actions

- FF also introduced **helpful actions**
  - Approximately:
    - The actions in the *first action level* of the delete-relaxed plan
    - Plus all *other* actions that could achieve the *same subgoals* (but did not happen to be chosen)
  - Helpful actions are more likely to lead you closer to the goal
    - Though they are not the only ones that could do so…
    - Slightly misleading name
  - FF restricts EHC to only use and apply helpful actions!

FF: Helpful Actions (2)

Suppose this is the relaxed plan generated by FF...



state-level 0 | action-level 1 | state-level 1 | …more levels

state-level 0: garbage, clean, quiet, ¬dinner, ¬present

action-level 1: carry, **roll**, **cook**, wrap

state-level 1: garbage, **¬garbage**, clean, ¬clean, quiet, ¬quiet, **dinner**, ¬dinner, present, ¬present

Possible in level 1:
carry, roll, cook, wrap

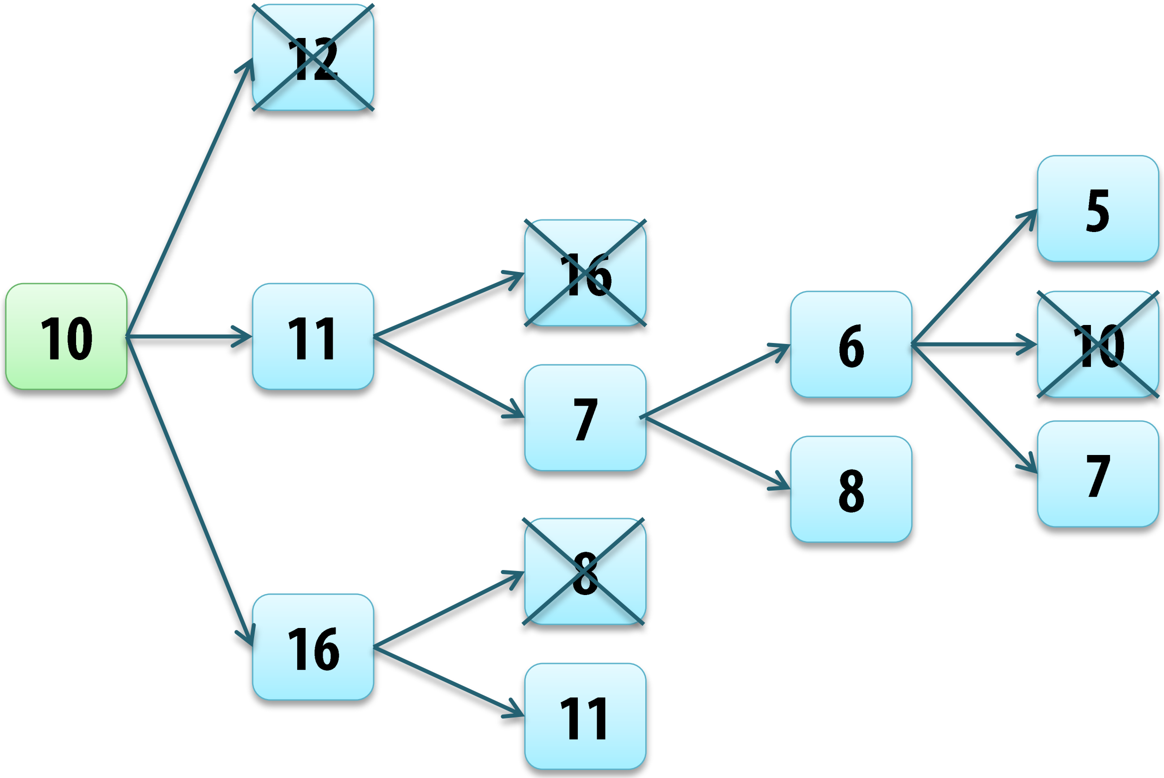
Used in relaxed plan:
roll, cook

Achieved at level 1:
¬garbage, dinner

Helpful actions:
carry, roll, cook
(because carry *could* have been used to achieve ¬garbage)

FF: EHC with Helpful Actions

- **<u>EHC</u>** with **<u>helpful actions</u>**:
  - 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
```

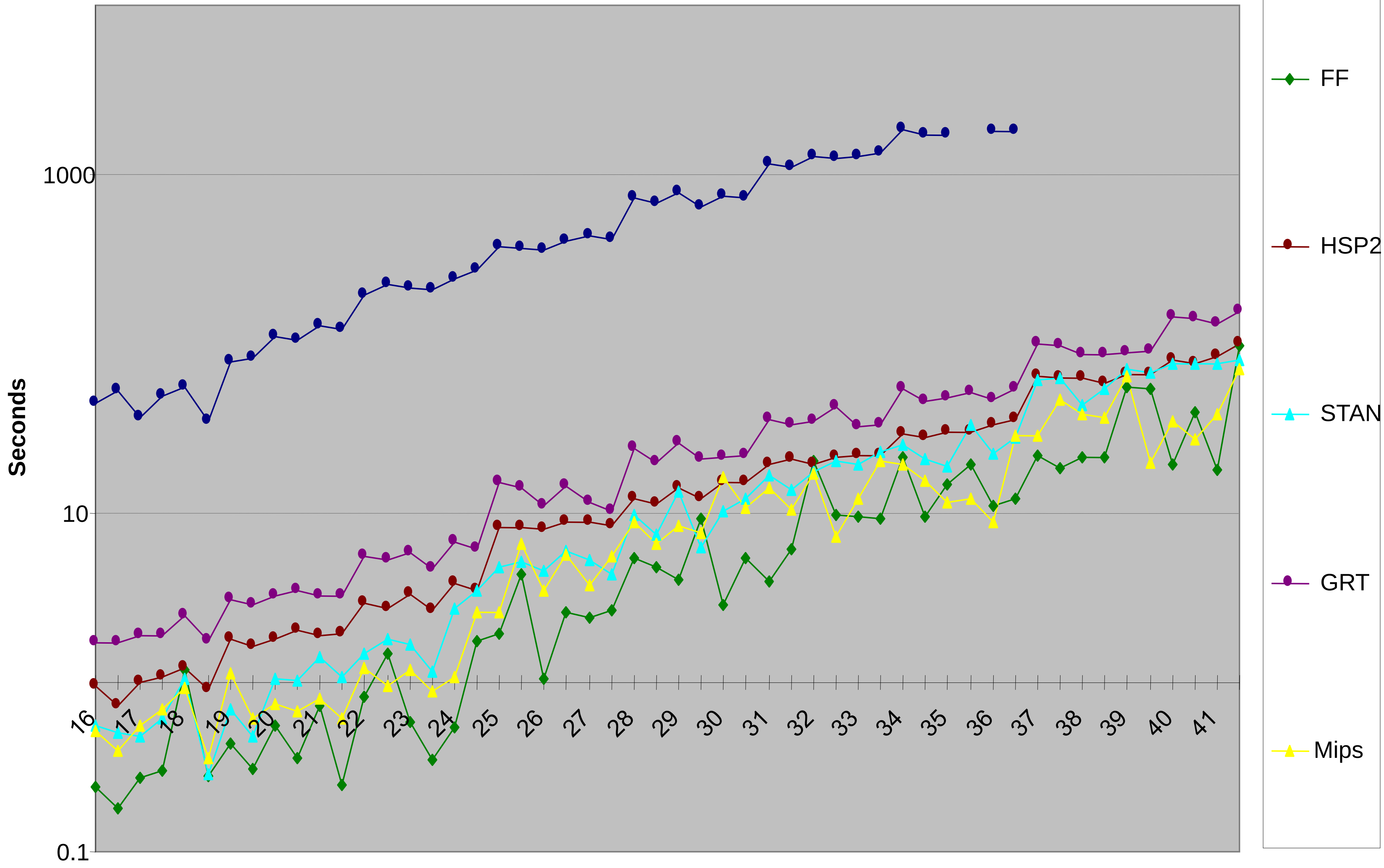
Incomplete
if there are dead ends!

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

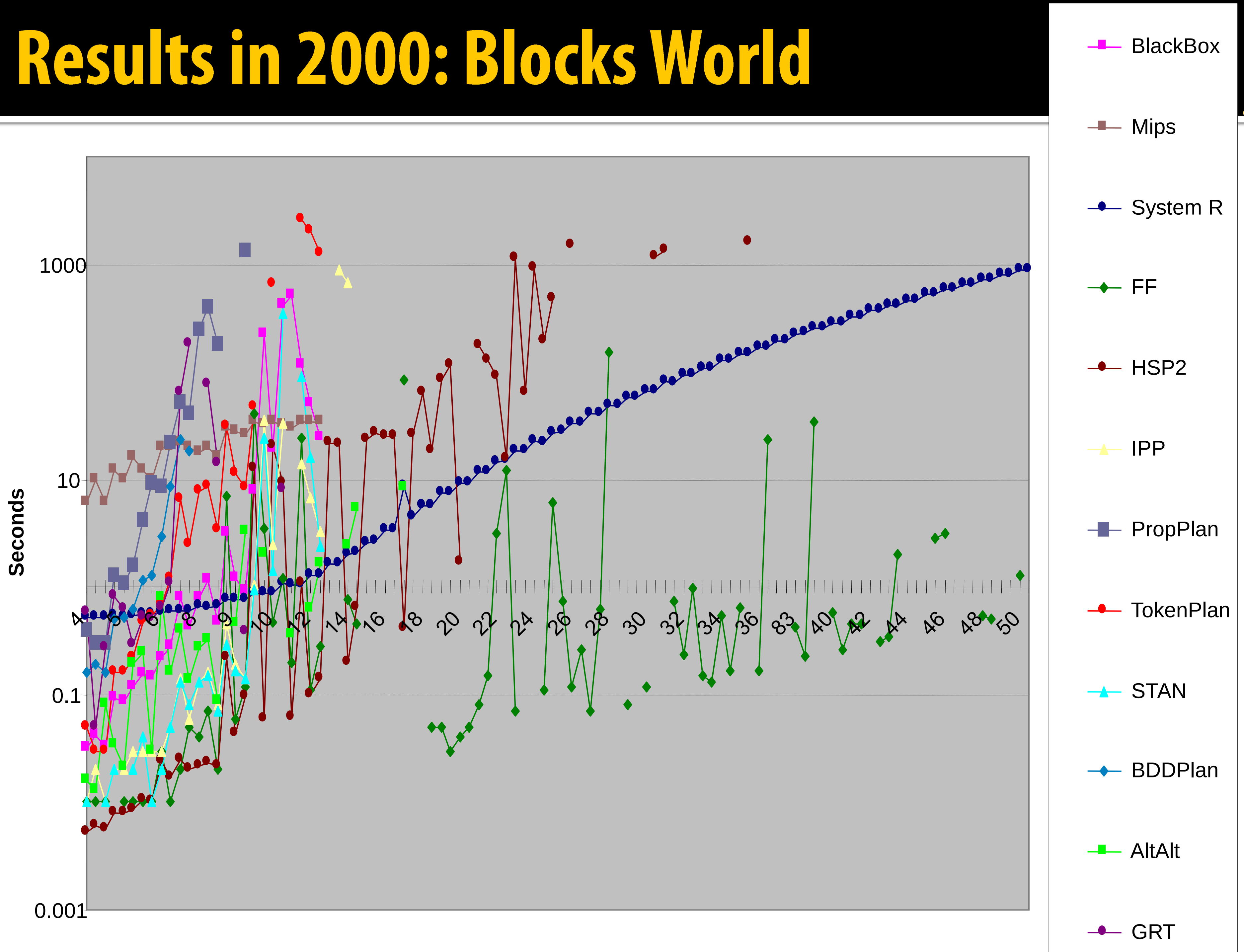
FF: Goal Ordering

- Finally:
  - During enforced hill climbing
    FF uses several goal ordering techniques
  - Like the use of EHC and helpful actions,
    these techniques also introduce incompleteness
    - For more information:
      Hoffmann & Nebel
      The FF Planning System: Fast Plan Generation through Heuristic Search

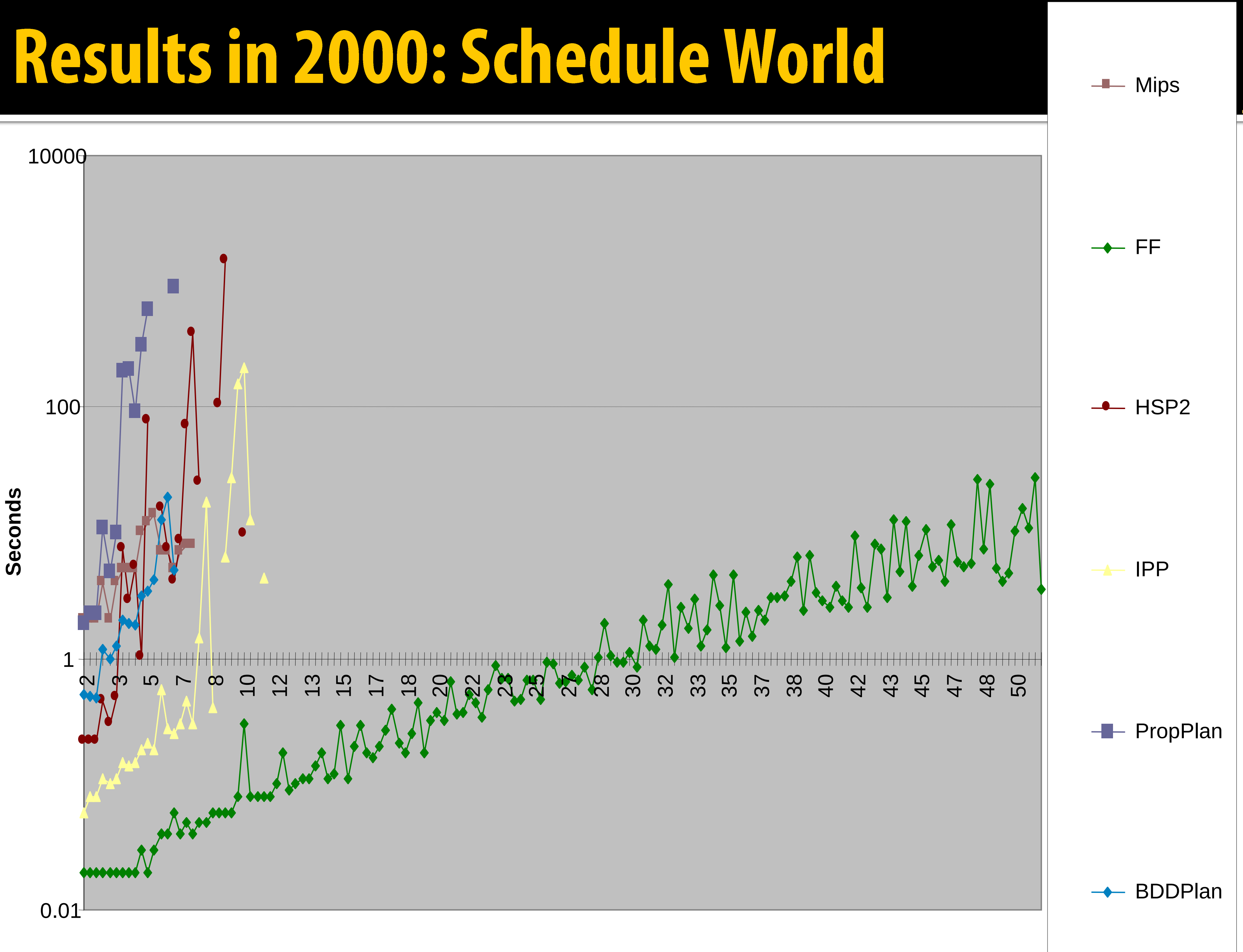
Results in 2000: Logistics



Results in 2000: Blocks World



Results in 2000: Schedule World



FF: Lasting Impact

- Lasting impact!
  - Many planners at least partly use the **FF heuristic**
    - At least as one of their possible heuristics
  - Many planners use **enforced hill climbing**
    - At least as one of their possible search methods
  - Many planners have **extended the relaxed planning graph**
    - For temporal actions, resources, …

Temporal Planning Graph

- Example: Temporal Planning Graph