Chapter 10: Classical Planning

Dana S. Nau

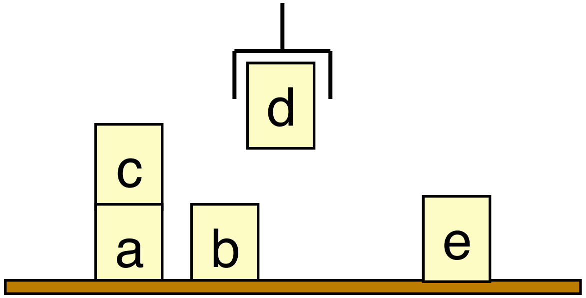
CMSC 421, Fall 2012

Motivation

- How to generate plans of action?
- Chapter 3: search algorithms
  - *Domain-independent* algorithms: work in many different problem domains
  - No standard representation for states of the world; needs domain-specific heuristics
- Chapter 7: logical agent for the wumpus world
  - Can develop domain-independent heuristics for manipulating logical formulas
  - Huge number of logical rules; can take forever to evaluate them if there are many actions and states
- Chapter 10: classical planning:
  - Standard representation of states and actions
  - Domain-independent algorithms and heuristics

Example: The Blocks World

- Infinitely wide table, finite number of children's blocks
- A robot hand that can pick up blocks and put them down
- A block can sit on the table or on another block
- Ignore where the blocks are located on the table
- Just consider
  - whether each block is on the table, on another block, or being held
  - whether each block is clear or covered by another block
  - whether the robot hand is holding anything
- Example state of the world:

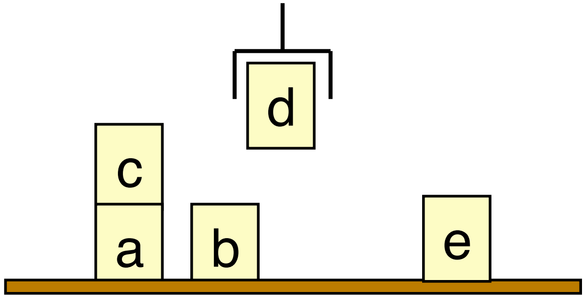


- Sounds trivial, but the search space can be very large
  - For *n* blocks, more than *n*! states

Symbols

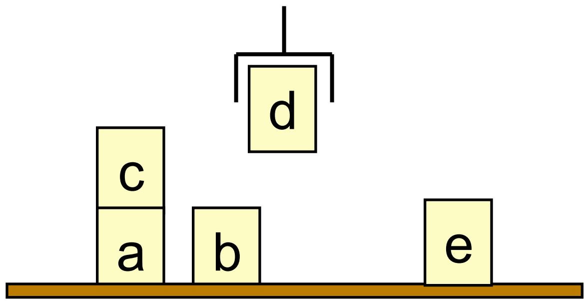
- Start with a first-order language
  - » Language of first-order logic
  - ◆ Restrict it to be *function-free*
    - » Finitely many predicate symbols and constant symbols,
    - » Unlimited (potentially infinite) set of variable symbols
    - » *No* function symbols

- Add a finite set of *operator names*
  - ◆ I'll discuss those later



Symbols for the Blocks World

- Constant symbols:
  - The blocks: a, b, c, d, e
- Predicates:
  - ontable(*x*) - block *x* is on the table
  - on(*x*,*y*) - block *x* is on block *y*
  - clear(*x*) - block *x* has nothing on it
  - holding(*x*) - the robot hand is holding block *x*
  - handempty - the robot hand isn't holding anything

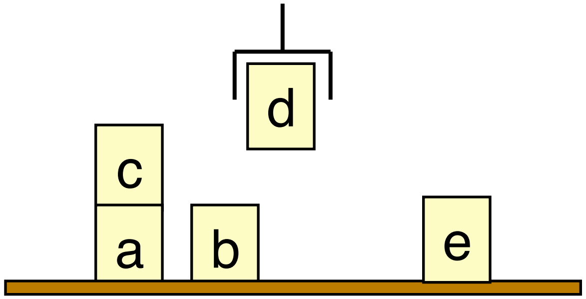


- Some terminology
  - *Atom*: predicate symbol and args
  - *Ground* expression: contains no variable symbols - e.g., on(c,a)
  - *Unground* expression: at least one variable symbol - e.g., on(c,*x*)

States

- State: a set *s* of ground atoms representing what's currently true

- Example:
  {ontable(a), on(c,a), clear(c),
  ontable(b), clear(b), holding(d),
  ontable(e), clear(e)}



- Number of possible states is finite
  - Suppose there are *c* constant symbols
  - *p* predicate symbols, each with *k* args
  - Then:
    - » Number of possible ground atoms is pc^k
    - » Number of possible states is 2^{pc^k}

Classical Operators

- *Operator*: a triple (head, preconditions, effects)
  - head: an operator name and a parameter list
    - E.g., $opname(x_1, \ldots, x_k)$
    - No two operators can have the same name
    - Parameter list must include *all* of the operator's variables
  - preconditions: literals that must be true to use the operator
  - effects: literals that the operator will make true

- We'll generally write operators in the following form:

  - $opname(x_1, \ldots, x_k)$
    - Precond: $p_1, p_2, \ldots, p_m$
    - Effects: $e_1, e_2, \ldots, e_n$

Blocks-World Operators

unstack(x,y)

Precond: on(x,y), clear(x), handempty

Effects: $\neg$on(x,y), $\neg$clear(x), $\neg$handempty, holding(x), clear(y)

stack(x,y)

Precond: holding(x), clear(y)

Effects: $\neg$holding(x), $\neg$clear(y), on(x,y), clear(x), handempty

pickup(x)

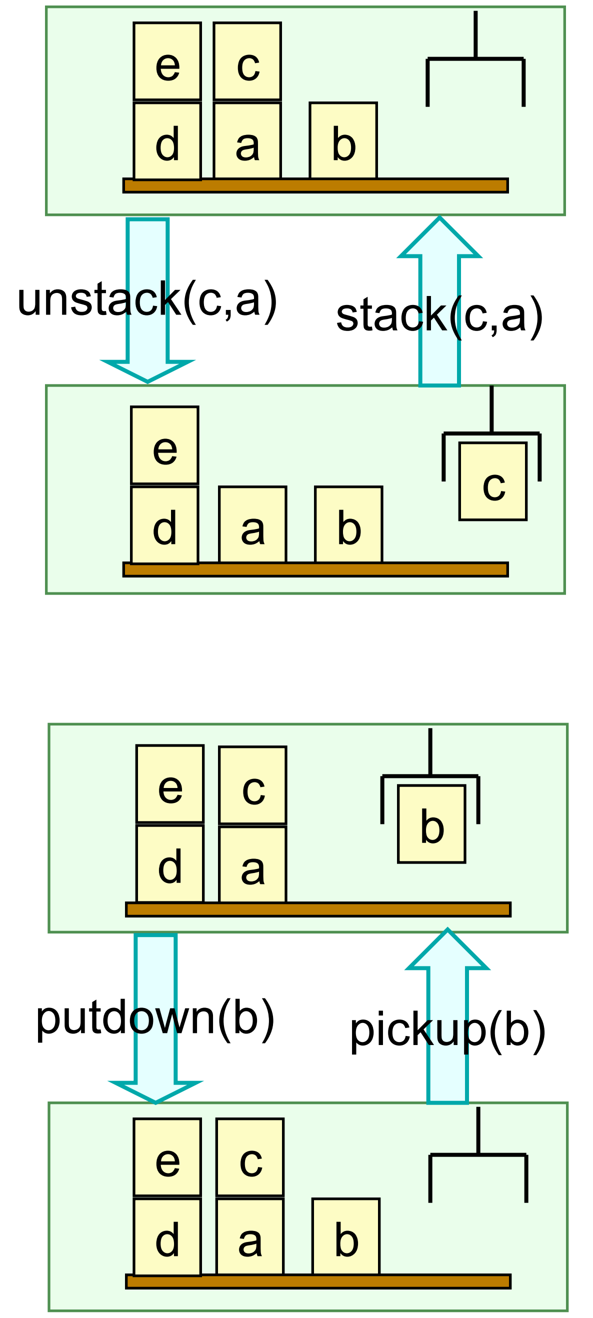
Precond: ontable(x), clear(x), handempty

Effects: $\neg$ontable(x), $\neg$clear(x), $\neg$handempty, holding(x)

putdown(x)

Precond: holding(x)

Effects: $\neg$holding(x), ontable(x), clear(x), handempty

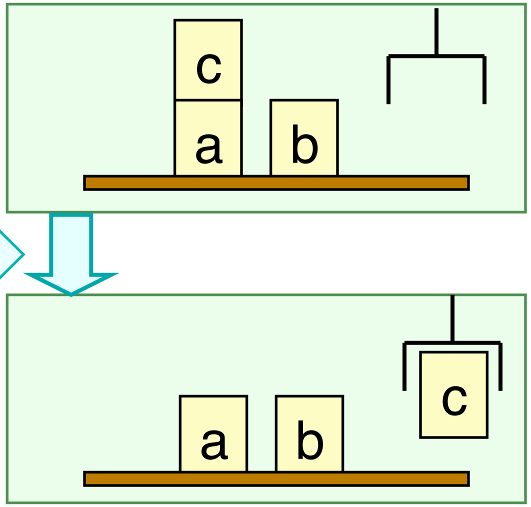


Actions and Plans

- Action: a ground instance (via substitution) of an operator

unstack(*x*,*y*)
Precond: on(*x*,*y*), clear(*x*), handempty
Effects: ¬on(*x*,*y*), ¬clear(*x*), ¬handempty,
holding(*x*), clear(*y*)

unstack(c,a)
Precond: on(c,a), clear(c), handempty
Effects: ¬on(c,a), ¬clear(c), ¬handempty,
holding(c), clear(a)



Notation

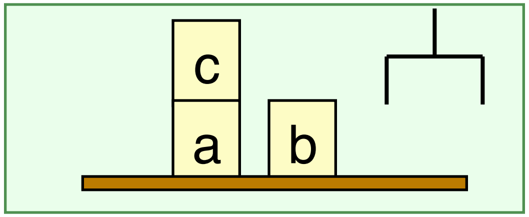
- Let S be a set of literals. Then
  - S^+ = {atoms that appear positively in S}
  - S^- = {atoms that appear negatively in S}
- Let a be an operator or action. Then
  - $\text{precond}^+(a)$ = {atoms that appear positively in precond(a)}
  - $\text{precond}^-(a)$ = {atoms that appear negatively in precond(a)}
  - $\text{effects}^+(a)$ = {atoms that appear positively in effects(a)}
  - $\text{effects}^-(a)$ = {atoms that appear negatively in effects(a)}
- Example:

  unstack(x,y)
  Precond: on(x,y), clear(x), handempty
  Effects: ¬on(x,y), ¬clear(x), ¬handempty,
  holding(x), clear(y)

  - effects^+(unstack(x,y)) = {holding(x), clear(y)}
  - effects^-(unstack(x,y)) = {on(x,y), clear(x), handempty}

Executability

- An action a is *executable* in s if s satisfies precond(a),
  - i.e., if $\text{precond}^{+}(a) \subseteq s$ and $\text{precond}^{-}(a) \cap s = \varnothing$
- An operator o is *applicable* to s if there is a ground instance a of o that is executable in s
- Example:
  - {ontable(a), on(c,a), clear(c), ontable(b), handempty}



unstack(x,y)
Precond: on(x,y), clear(x), handempty
Effects: ¬on(x,y), ¬clear(x), ¬handempty, holding(x), clear(y)

unstack(c,a)
Precond: on(c,a), clear(c), handempty
Effects: ¬on(c,a), ¬clear(c), ¬handempty, holding(c), clear(a)

Performing an Action

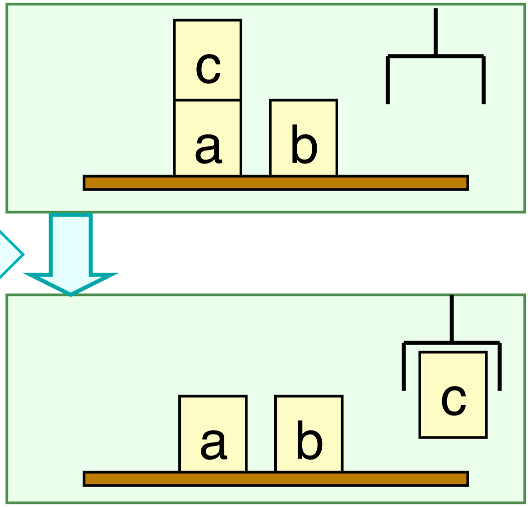
- If a is executable in s, the result of performing it is

$$\gamma(s,a) = (s - \text{effects}^{-}(a)) \cup \text{effects}^{+}(a)$$

  - Delete the negative effects, and add the positive ones
- Example:

  s = {ontable(a), on(c,a), clear(c), ontable(b), handempty}

  a = unstack(c,a)

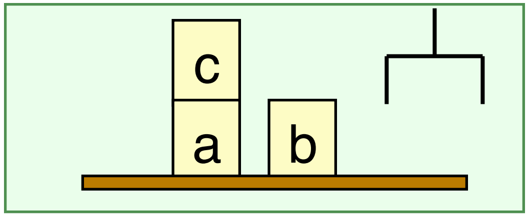


unstack(c,a)
  Precond: on(c,a), clear(c), handempty
  Effects: ¬on(c,a), ¬clear(c), ¬handempty,
  holding(c), clear(a)

- $\gamma(s,a)$ = {ontable(a), ~~on(c,a), clear(c),~~ ontable(b), clear(b), ~~handempty,~~ holding(c), clear(a)}
  - The book calls this Result(s,a)

Executability of Plans

- Plan: a sequence of actions $\pi = (a_1, \ldots, a_n)$
- A plan $\pi = (a_1, \ldots, a_n)$ is *executable* in the state s_0 if
  - » a_1 is executable in s_0, producing some state $s_1 = \gamma(s_0, a_1)$
  - » a_2 is executable in s_1, producing some state $s_2 = \gamma(s_1, a_2)$
  - » …
  - » a_n is executable in s_{n-1}, producing some state $s_n = \gamma(s_{n-1}, a_n)$
- In this case, we define $\gamma(s_0, \pi) = s_n$
- Example on next slide



s = {ontable(a), on(c,a), clear(c), ontable(b),clear(b), handempty}

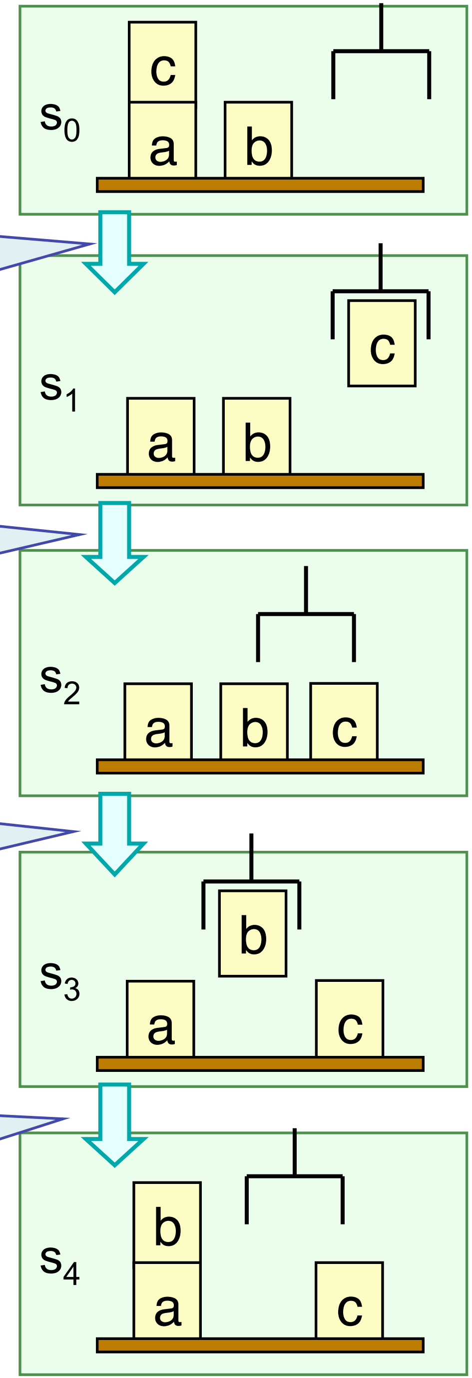
π = (unstack(c,a), putdown(c), pickup(b), stack(b,a))

unstack(c,a)
Precond: on(c,a), clear(c), handempty
Effects: ¬on(c,a), ¬clear(c), ¬handempty, holding(c), clear(a)

putdown(c)
Precond: holding(c)
Effects: ¬holding(c), ontable(c), clear(c), handempty

pickup(b)
Precond: ontable(b), clear(b), handempty
Effects: ¬ontable(b), ¬clear(b), ¬handempty, holding(b)

stack(b,a)
Precond: holding(b), clear(a)
Effects: ¬holding(b), ¬clear(a), on(b,a), clear(b), handempty



Problems and Solutions

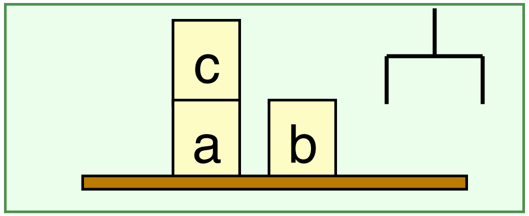
- *Planning problem*: a triple $P = (O, s_0, g)$
  - O is a set of operators
  - s_0 is the *initial state* - a set of atoms
  - g is the *goal formula* - a set of literals
- Every state that satisfies g is a *goal state*

- A plan π is a *solution* for $P=(O,s_0,g)$ if
  - π is executable in s_0
  - the resulting state $\gamma(s_0,\pi)$ satisfies g

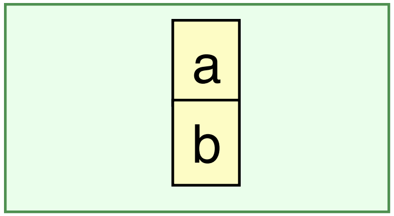
Example

- O = {stack(x,y), unstack(x,y), pickup(x), putdown(x)}

- s_0 = {ontable(a), on(c,a), clear(c),
  ontable(b), clear(b), handempty}



- g = {on(a,b)}



- One of the solutions is
  - π = (unstack(c,a), putdown(c), pickup(a), stack(a,b))

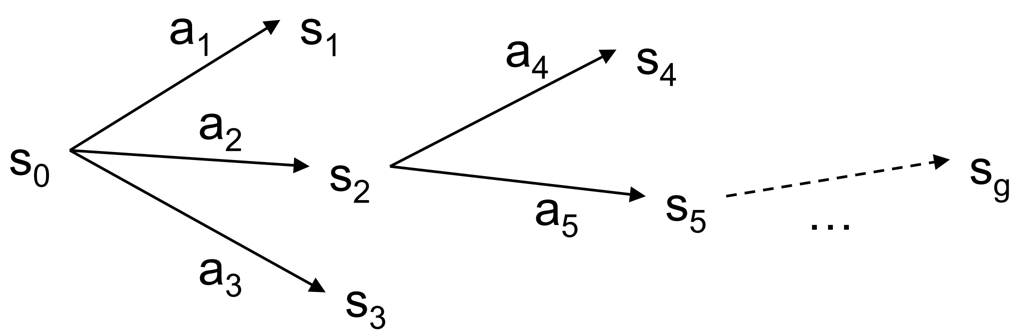
Complexity of Planning

- Given a classical planning problem *P*, does it have a solution?
  - PSPACE-complete (much harder than NP-complete)
- Given a classical planning problem *P* and an integer *k*, is there a solution of length *k* or less?
  - Again PSPACE-complete

- Suppose we add function symbols to the language
- Given a planning problem *P*, does it have a solution?
  - Undecidable
- Given a planning problem *P* and an integer *k*, is there a solution of length *k* or less?
  - Decidable, NEXPTIME-complete

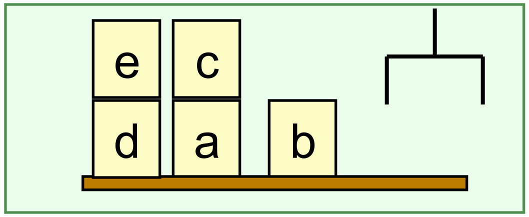
Forward Search

- Go forward from the initial state



- Breadth-first and best-first
  - *Sound*: if they return a plan, then the plan is a solution
  - *Complete*: if a problem has a solution, then they will return one
  - Usually not practical because they require too much memory
    - » Memory requirement is exponential in the length of the solution
- Depth-first search, greedy search
  - More practical to use
  - Worst-case memory requirement is linear in the length of the solution
  - Sound but not complete
- But classical planning has only finitely many states
  - Thus, can make depth-first search complete by doing loop-checking
- The book also discusses backward search, but I'll skip it

Reducing Search Space Size



- Suppose there were 450 blocks rather than 5
- Search space size is more than 10^{1000}
  - Most of the states are completely irrelevant for whatever goal we might want to achieve
  - A search algorithm might waste time trying many of them
- How to reduce the size of the search space?
- One approach:
  - 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

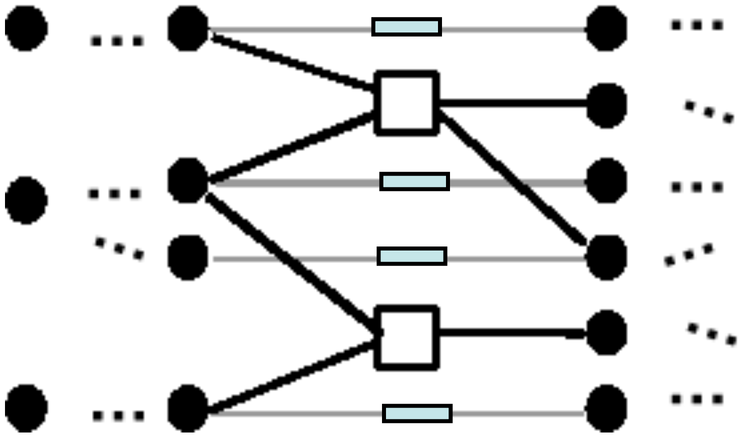
Graphplan

procedure Graphplan:

- for $k = 0, 1, 2, \ldots$
  - *Graph expansion:*
    - create a "planning graph" that contains k "levels"
  - Check whether the planning graph satisfies a necessary (but insufficient) condition for plan existence

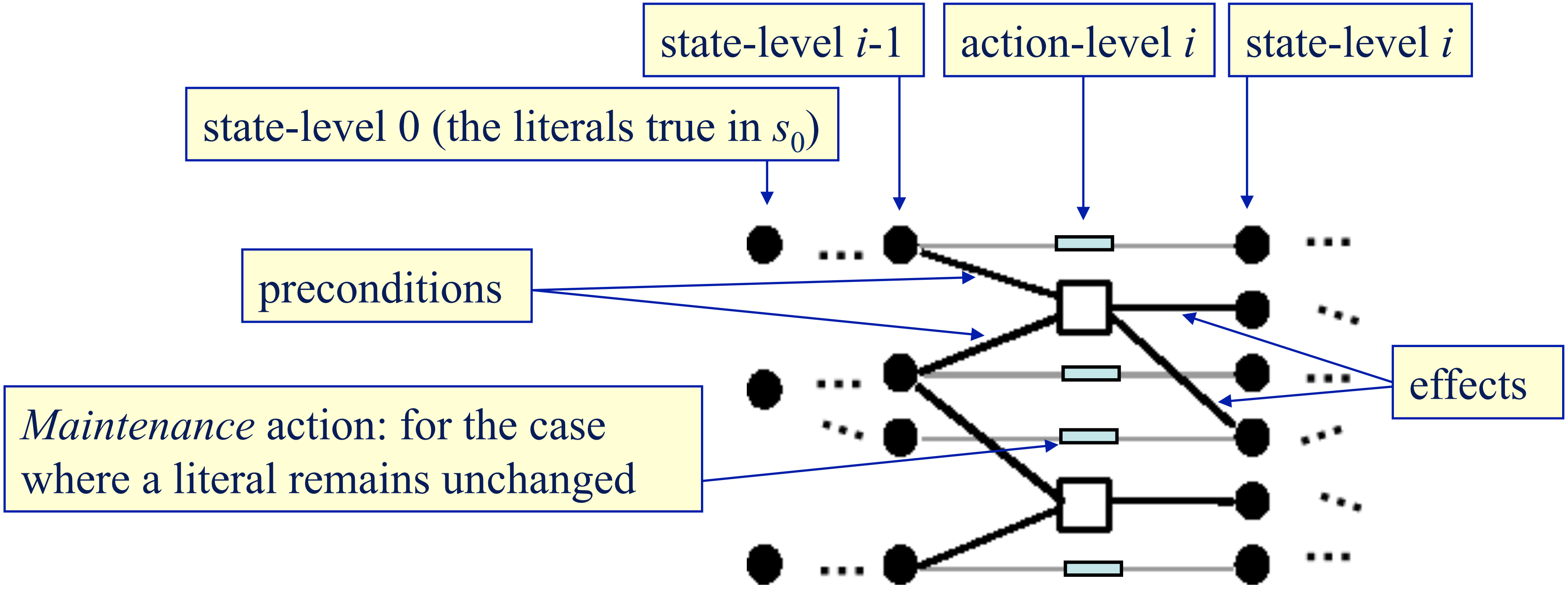
relaxed problem

  - 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 = {garbage, cleanHands, quiet}

  g = {dinner, present, ¬garbage}

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

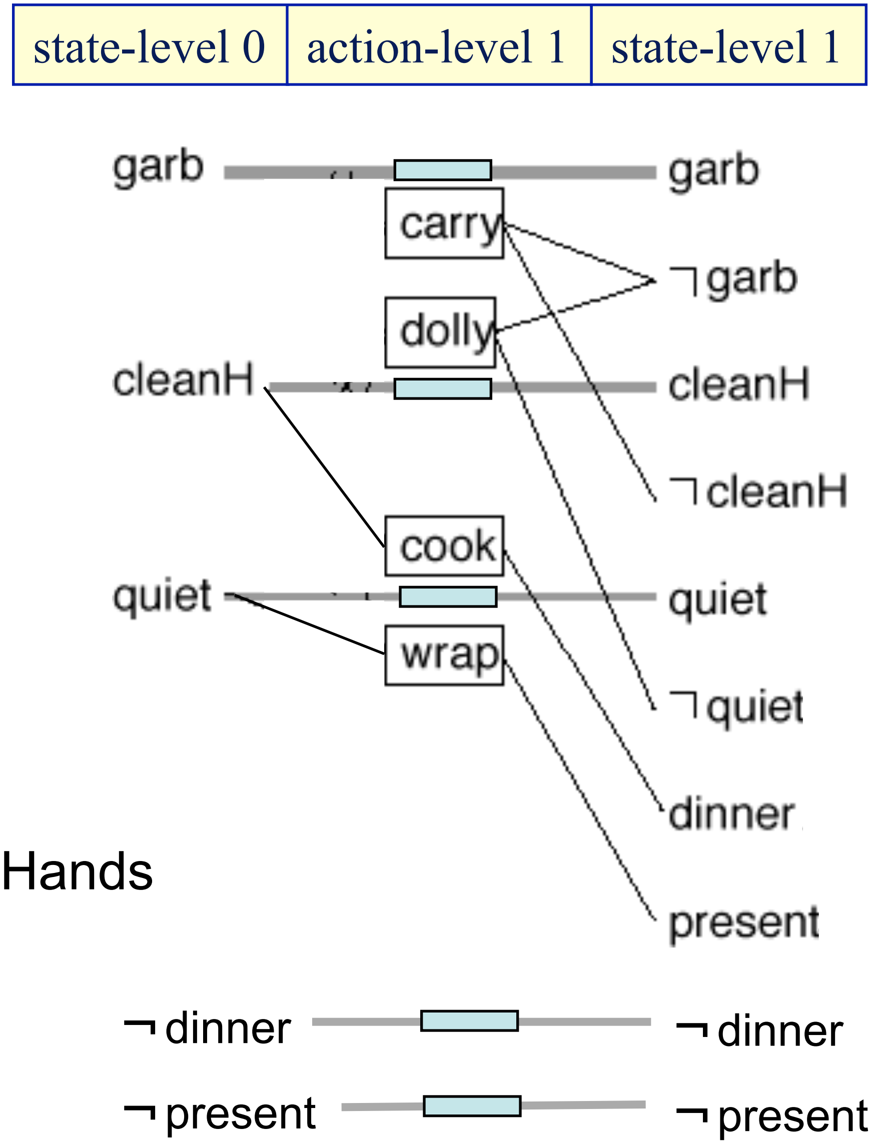
Also have the maintenance actions: one for each literal

Example (continued)

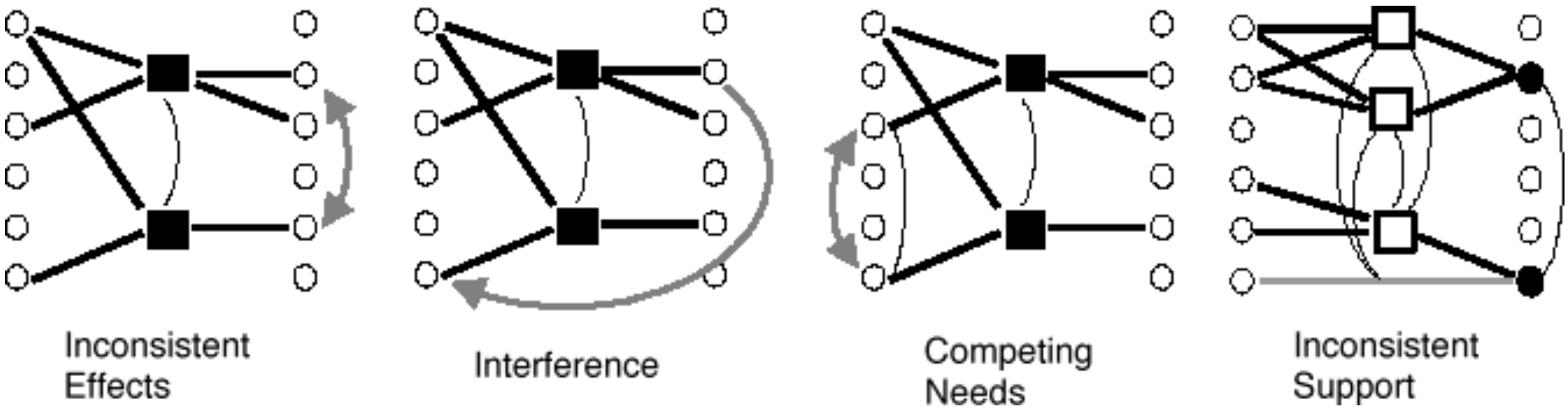
- state-level 0:
  {all atoms in s_0} U
  {negations of all atoms not in s_0}
- action-level 1:
  {all actions whose preconditions
  are satisfied and non-mutex in s_0}
- state-level 1:
  {all effects of all of the
  actions in action-level 1}

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

Also have the maintenance actions



Mutual Exclusion

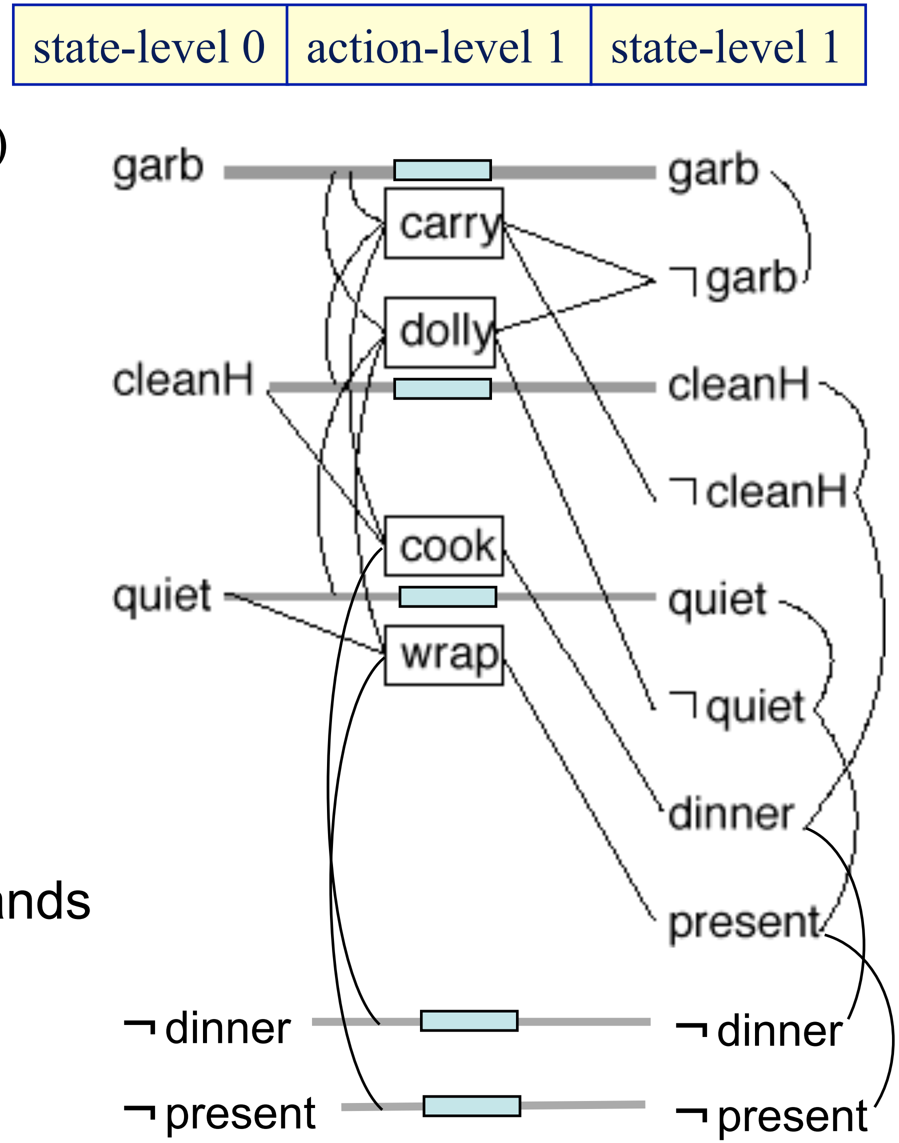


- Two actions at the same action-level are mutex if
  - *Inconsistent effects:* an effect of one negates an effect of the other
  - *Interference:* one deletes a precondition of the other
  - *Competing needs:* **they have mutually exclusive preconditions**
- Otherwise they don't interfere with each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

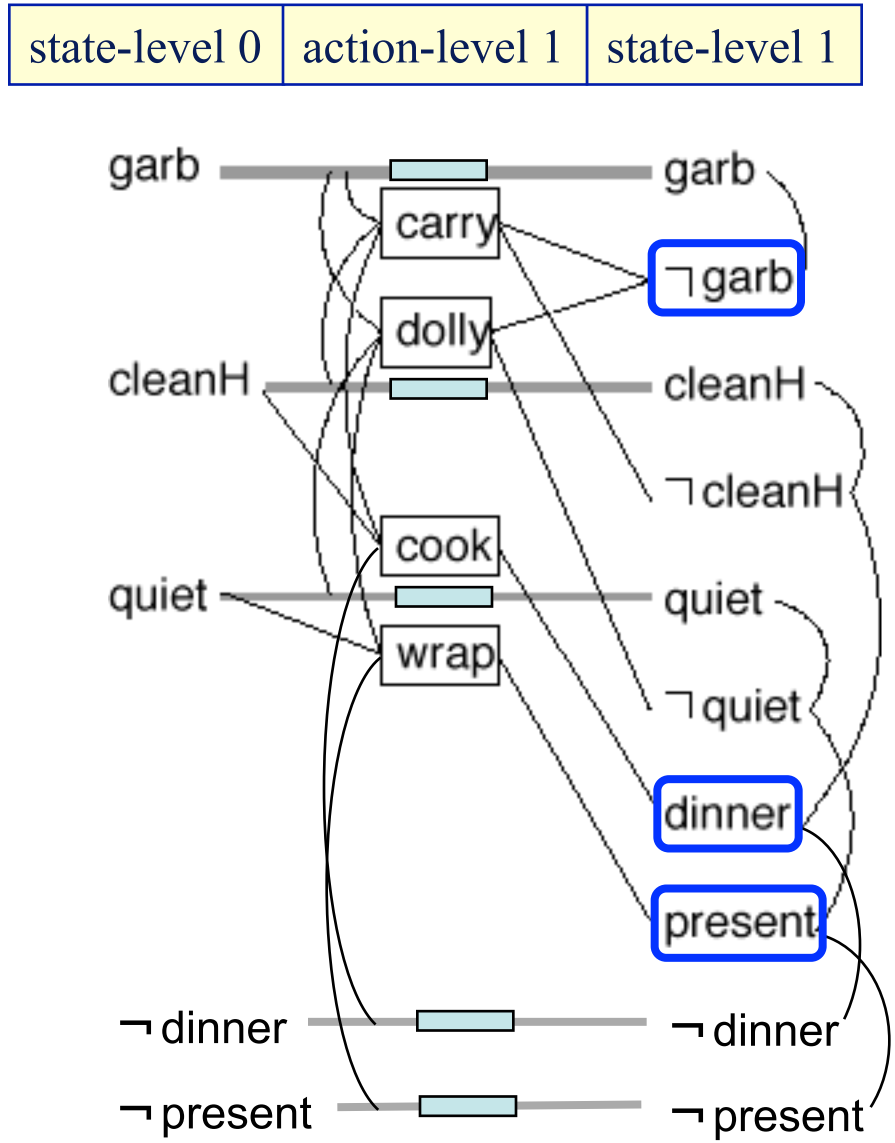


Action	Preconditions	Effects
cook()	cleanHands	dinner
wrap()	quiet	present
carry()	*none*	¬garbage, ¬cleanHands
dolly()	*none*	¬garbage, ¬quiet

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



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

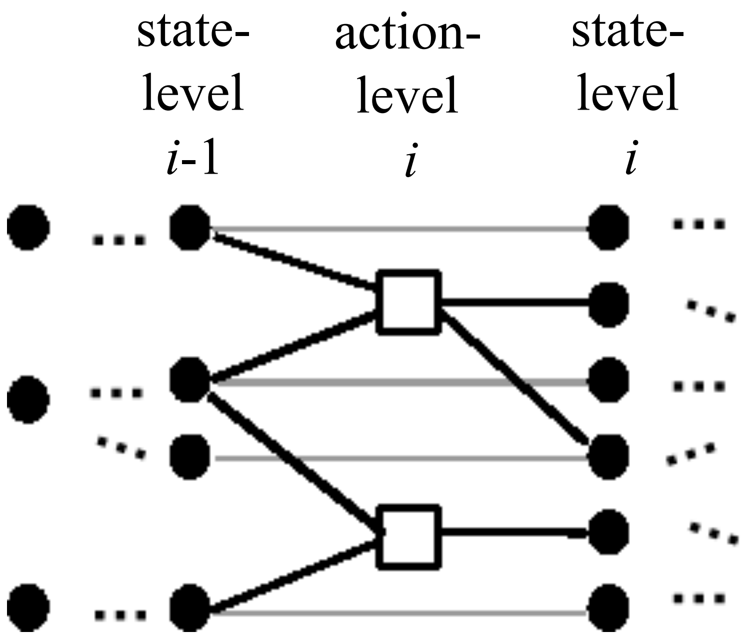
if any pair of chosen actions are mutex

then backtrack

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

Solution-extraction(g', j–1)

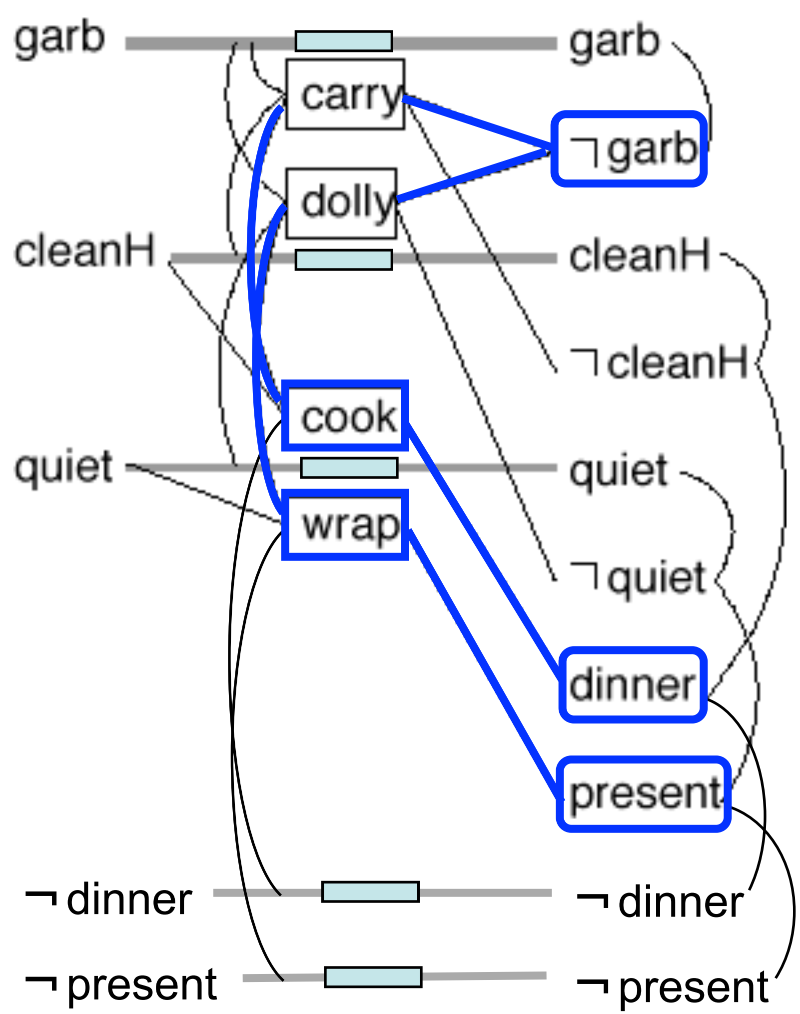
end Solution-extraction



Example (continued)

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

state-level 0	action-level 1	state-level 1



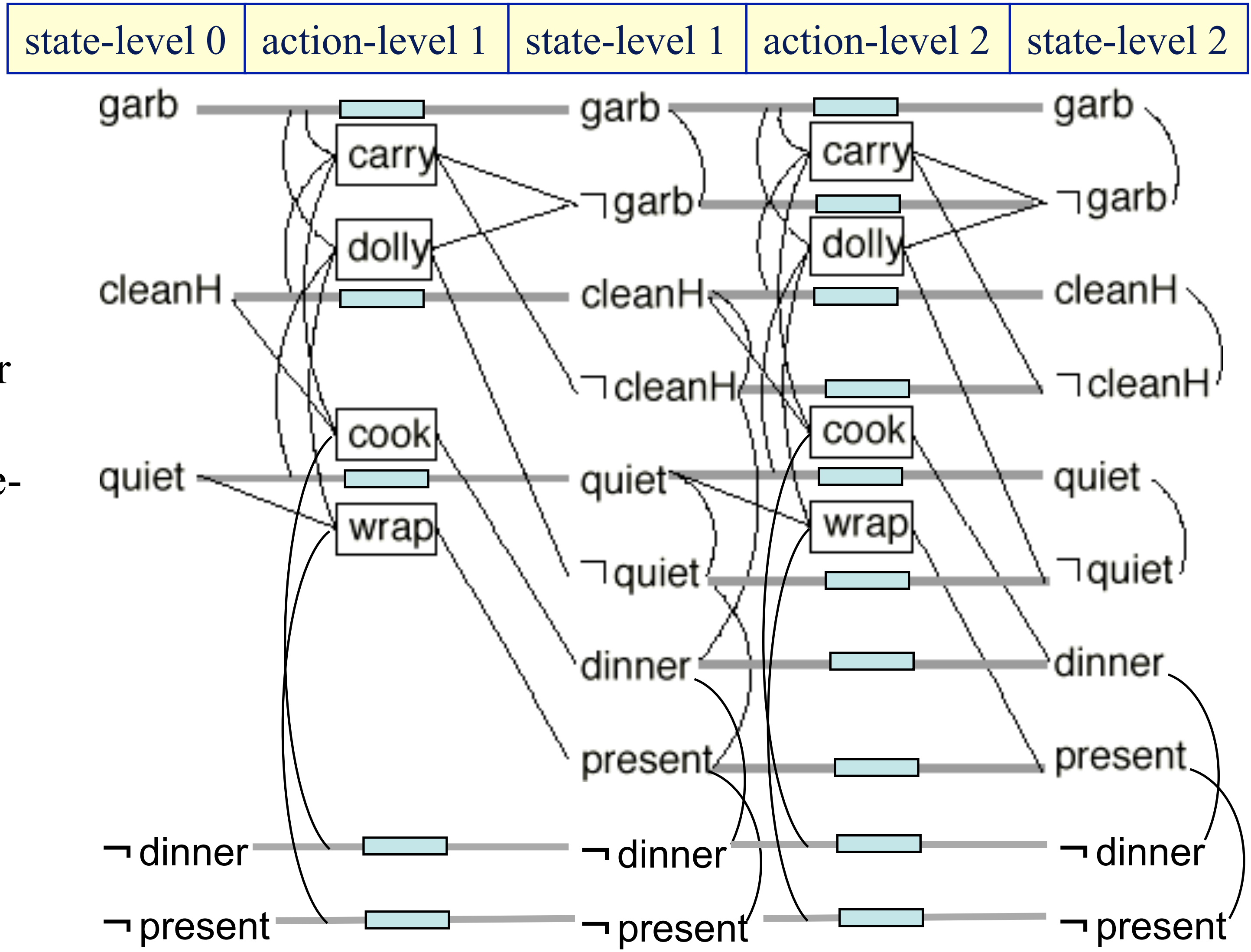
Recall what the algorithm does

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

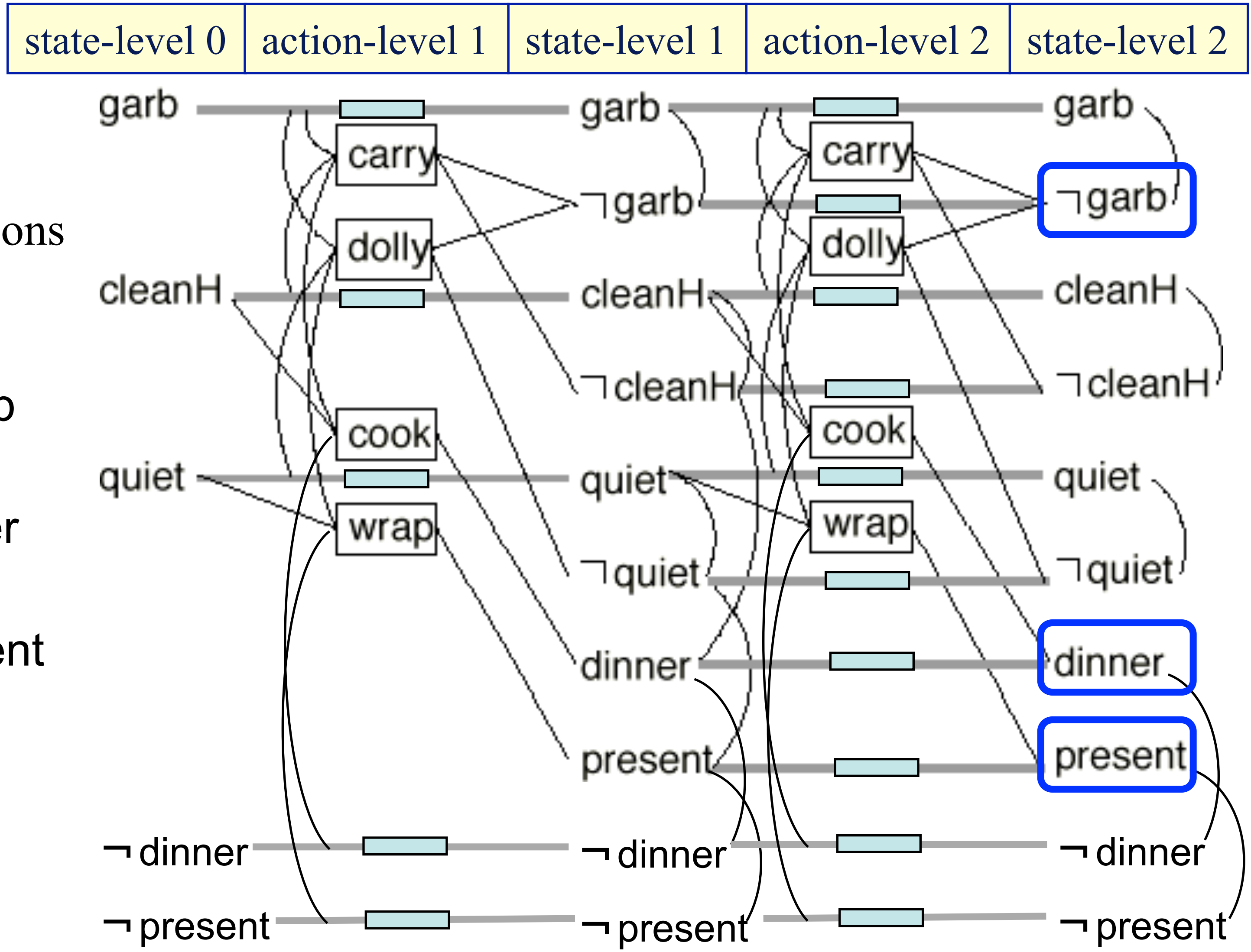
Example (continued)

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



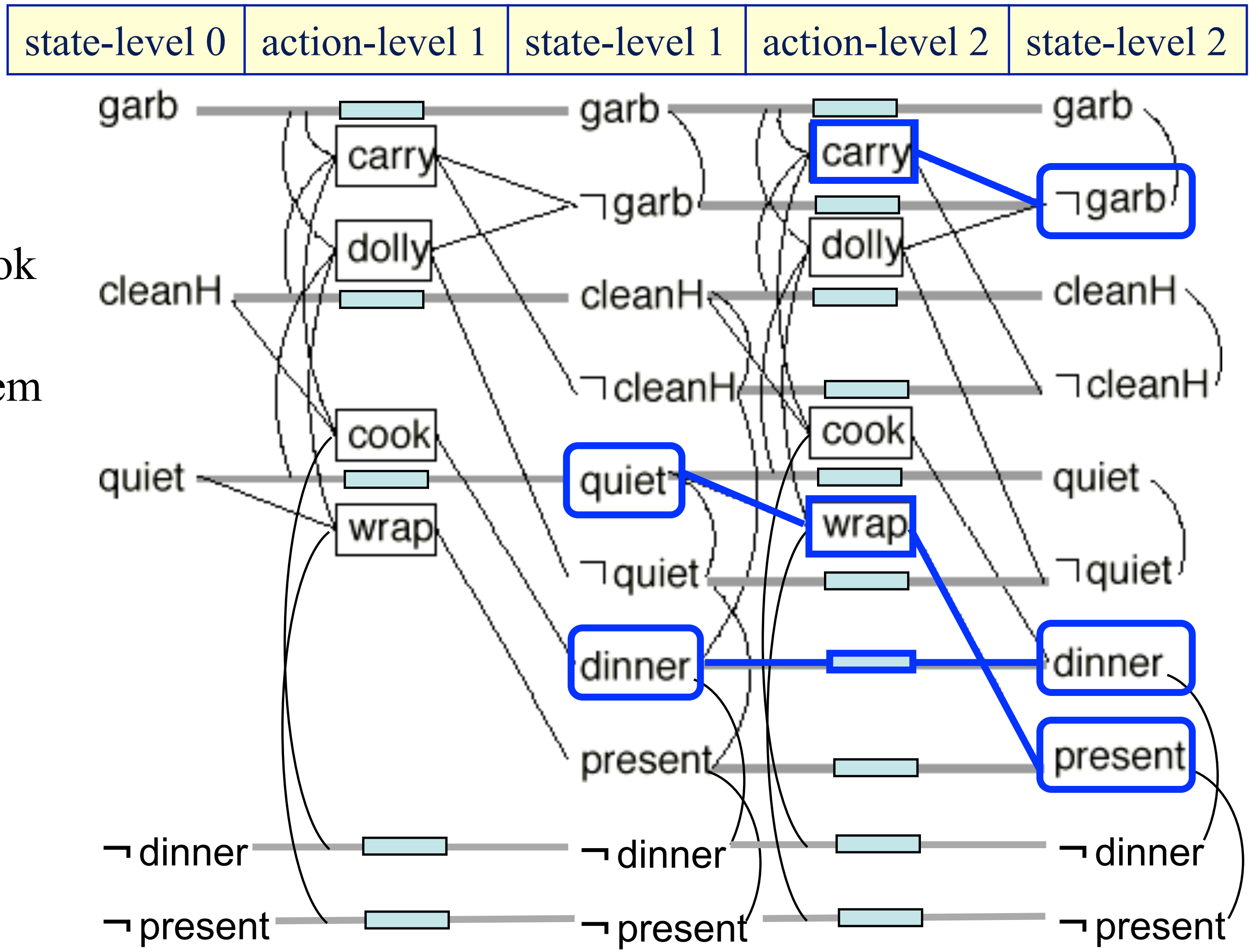
Example (continued)

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



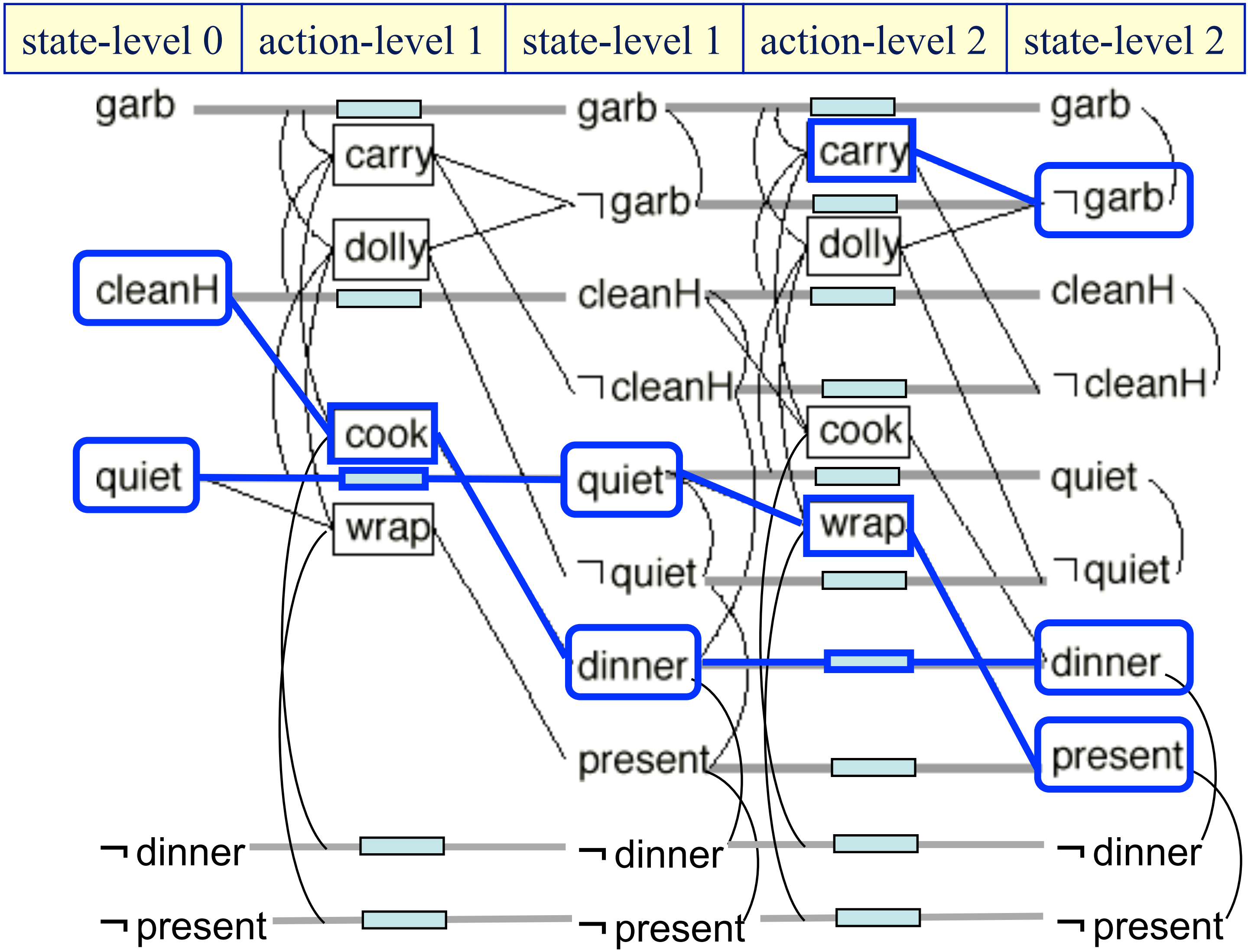
Example (continued)

- Several of the combinations look OK at level 2
- Here's one of them

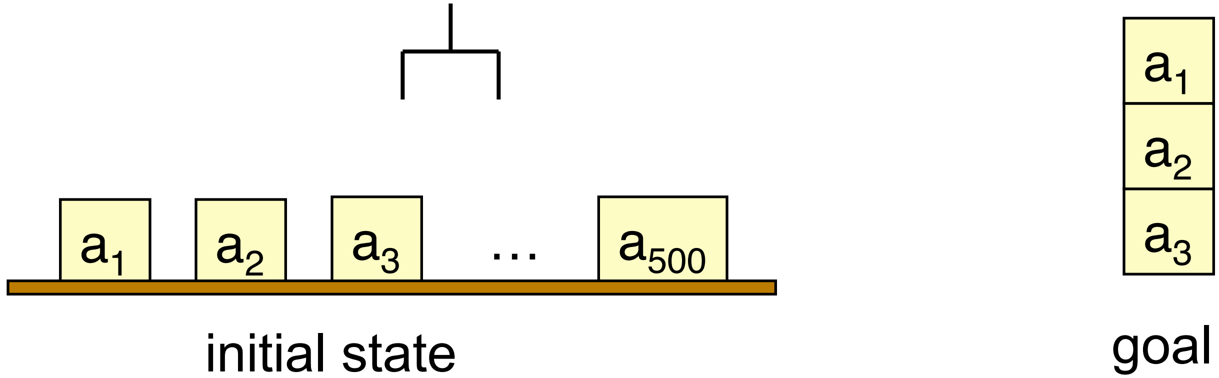


Example (continued)

- Call Solution-Extraction recursively at level 2
- It succeeds
- Solution whose *parallel length* is 2



Back to Forward Search



- Earlier, I said
  - Forward search can waste time trying lots of irrelevant actions (see above)
    - » pickup(a_1), pickup(a_2) , …, pickup(a_{500})
  - Need a good heuristic to guide the search

- We can use planning graphs to compute such a heuristic

Getting Heuristic Values from a Planning Graph

- Recall how GraphPlan works:

loop

Graph expansion: this takes polynomial time

extend a "planning graph" forward from the initial state until we have achieved a necessary (but insufficient) condition for plan existence

Solution extraction: this takes exponential time

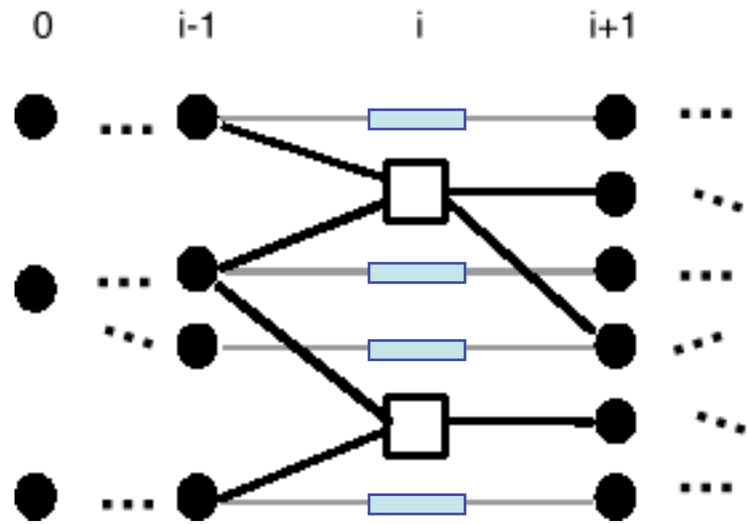
search backward from the goal, looking for a correct plan

if we find one, then return it

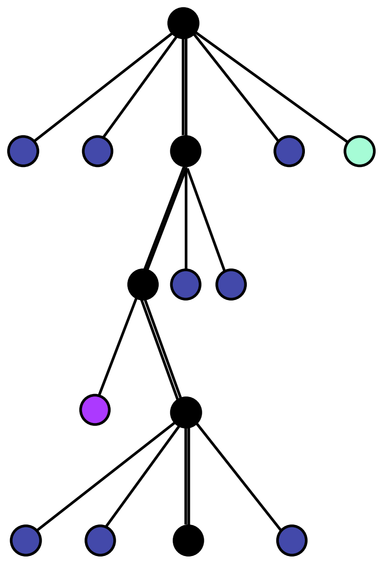
repeat

Using Planning Graphs to Compute *h*(*s*)

- In the graph, there are alternating layers of ground literals and actions
- The number of "action" layers is a lower bound on the number of actions in the plan
- Construct a planning graph, starting at *s*
- $\Delta^g(s,g)$ = level of the first layer that "possibly achieves" the goal
  - Some ways to improve this, but I'll skip the details



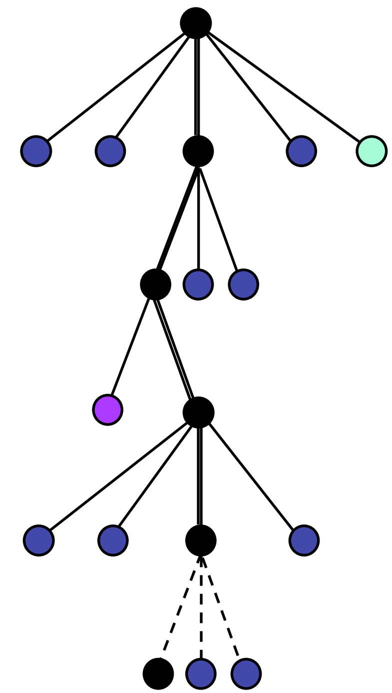
The FastForward Planner



- Use a heuristic function $h(s)$ similar to $\Delta^g(s,g)$
- Don't want an A*-style search (takes too much memory)
- Instead, use a greedy procedure:

until we have a solution, do
    expand the current state *s*
    s := the child of *s* for which *h(s)* is smallest
      (i.e., the child we think is closest to a solution)

The FastForward Planner



- Use a heuristic function $h(s)$ similar to $\Delta^g(s,g)$
- Don't want an A*-style search (takes too much memory)
- Instead, use a greedy procedure:

```
until we have a solution, do
        expand the current state s
        s := the child of s for which h(s) is smallest
           (i.e., the child we think is closest to a solution)
```

- Problem: can get caught in local minima
  - $h(s') > h(s)$ for every successor s' of s
  - Escape by doing a breadth-first search until you find a node with lower cost
- Problem: can hit a dead end - in this case, FF fails
- No guarantee on whether FF will find a solution, or how good a solution
  - But FF works quite well on many classical planning problems

International Planning Competitions

- International planning competitions in 1998, 2002, 2004, 2006, 2008
  - Many of the planners in these competitions have incorporated ideas from GraphPlan and FastForward
- Graphplan was developed in 1995
  - Several years before the competitions started
- FastForward was introduced in the 2000 International Planning Competition
  - It got one of the two top awards
  - Large variation in how good or bad its plans were, but it found them very quickly