Planning and Scheduling: Neoclassical Planning

Hochschule Bonn-Rhein-Sieg

b-it Bonn-Aachen International Center for Information Technology

Prof. Dr.-Ing. Gerhard K. Kraetzschmar

Neoclassical Planning

- Historical development
  - Classical planning seemed stalled: expressiveness, complexity
  - Revival of research on classical planning by neoclassical methods
- Main differences
  - Classical Planning:
    - Every node in search space is partial plan (action sequence, partial order set)
    - Every solution reachable from this state contains all actions of partial plan
  - Neoclassical Planning:
    - Every node in search space can be viewed as set of several partial plans
    - Not every action in a node appears in a solution plan reachable from node
- Techniques
  - Planning-graph techniques
  - Propositional satisfiability techniques
  - Constraint satisfaction techniques

Planning and Scheduling: Graph-Based Techniques

Hochschule Bonn-Rhein-Sieg

b-it Bonn-Aachen International Center for Information Technology

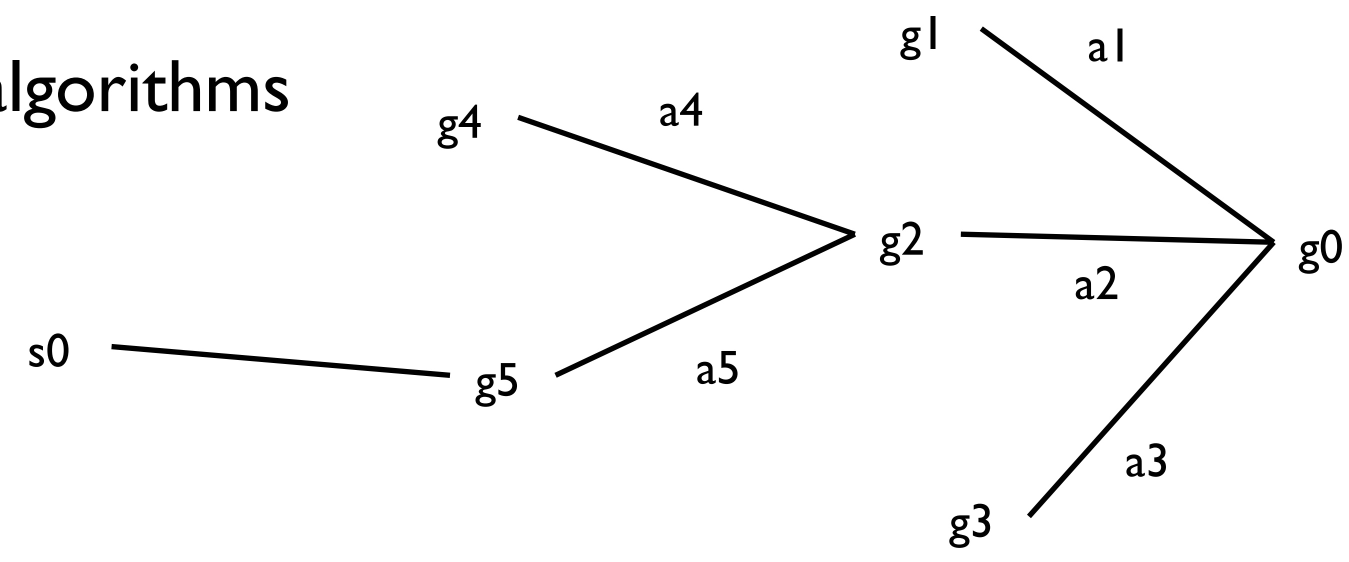
Prof. Dr.-Ing. Gerhard K. Kraetzschmar

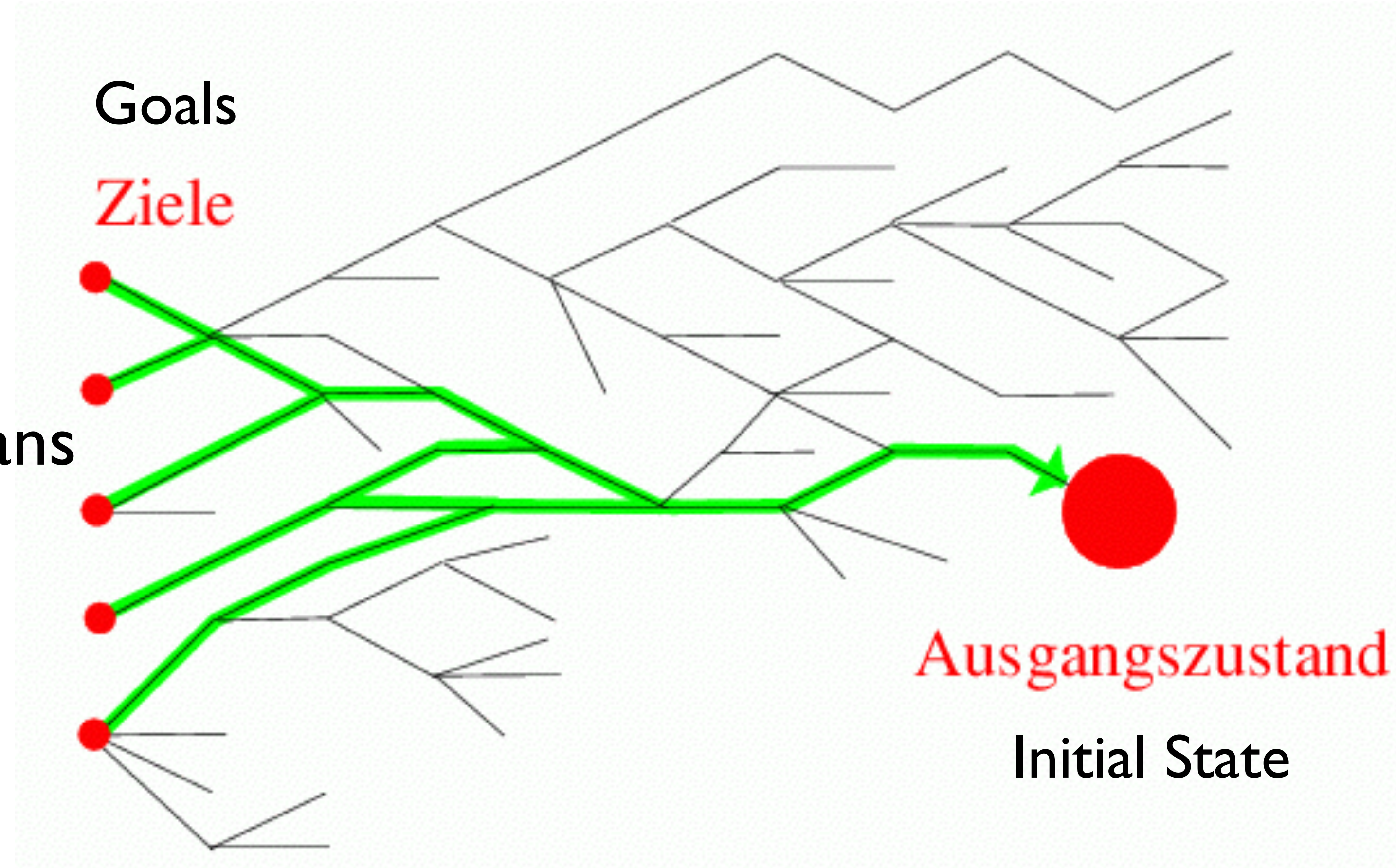
Acknowledgements

- These slides are based on those slides by
Dana Nau, Alan Fern, Jose Luis Ambite, Jim Blythe, and Dan Weld
- A lot of improvements have been made by Iman Awaad

Motivation

- A big source of inefficiency in search algorithms is the large branching factor.
  - E.g.: Forward search may try many actions that never reach a goal state.
  - E.g.: Backward search may try many actions that cannot be reached from S_0.





- POP searches in space of partial plans
- …backwards from goal state(s) to initial state(s)
- …in **unstructured** search space

Motivation (cont'd)

- To reduce branching factor:
  - First, create a relaxed problem
    - Remove some restrictions of the original problem
      (we want the relaxed problem to be easy to solve (polynomial time))
    - Solutions to the relaxed problem include all solutions to the original problem
      (but not all relaxed problem solutions are solutions of the original problem)
  - Then, do a modified version of the original search
    - Restrict the search space to include only those actions
      that occur in the solutions to the relaxed problem
- GraphPlan does this by building and then searching a special structure
  - This time, the output is a sequence of sets of actions:
    - For example, <{a1, a2}, {a3,a4},{a5, a6, a7}>
    - Execute the **sets of actions** in the specified order
    - Execute the **actions within** each set in **any order**

Motivation (cont'd)

- What have we taken advantage of in past techniques?
  - Notion of relevance
  - Divide and conquer
  - Goal dependence/independence
  - Partial instantiation (least commitment strategy)
  - Partial ordering (least commitment strategy)
  - Domain information
  - …
- What are we taking advantage of now?
  - Reachability
  - Dependencies between operators
- Flaws are not independent… and their resolvers may interfere
- We post dependencies as constraints that we deal with at some later stage

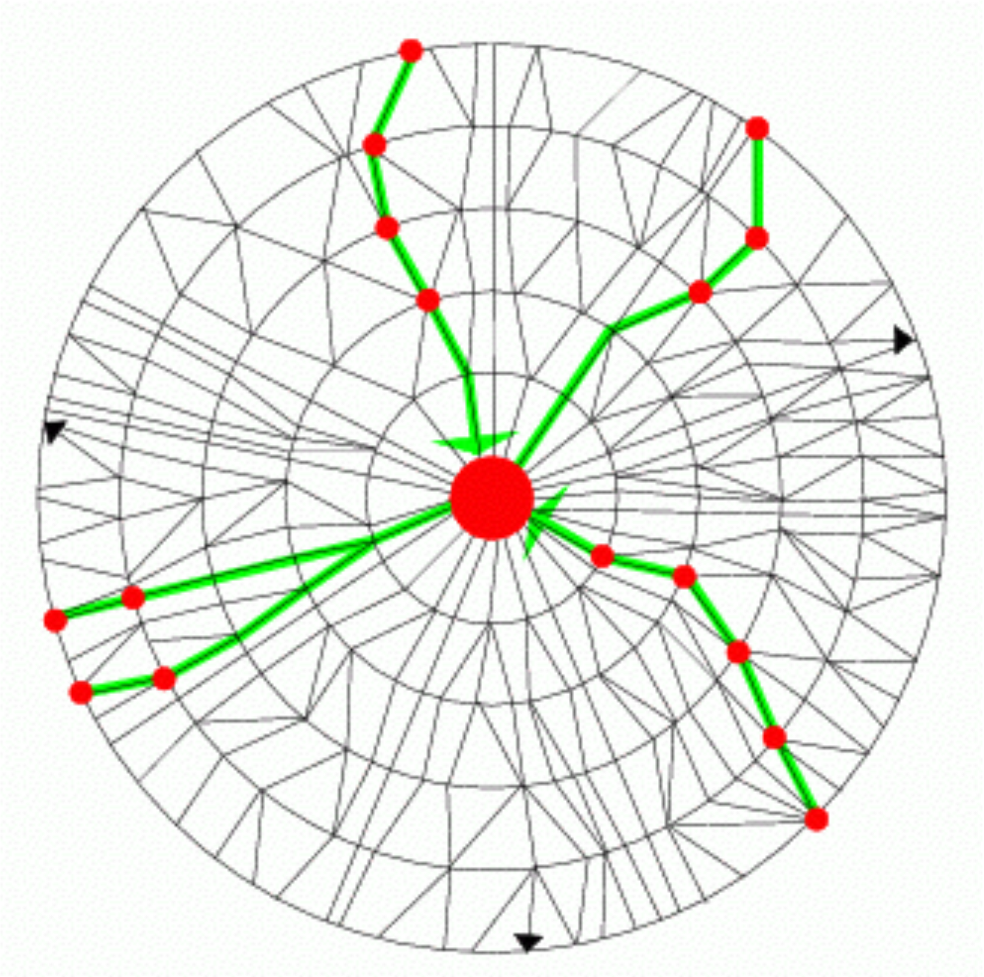
Dependencies of Plan Operators

- Plan operators may be mutually dependent in various ways
- Example:
  - **`Go`** generates and deletes preconditions for **`Deliver`** and **`Take`**
  - **`Deliver`** deletes effects of **`Take`**
- Actions are ground instances of operators
- Whether dependencies between planning operators actually occur can only be determined for ground instances (actions)
- The basis for improving planning algorithms is exploiting dependencies between planning operators

```
• Op(Action: Go(PlaceA, PlaceB)
     Pre: In(PlaceA)
     Eff: Add In(PlaceB)
          Del In(PlaceA)   )
• Op(Action: Deliver(Letter,Place)
     Pre: Have(Letter)
          Receiver(Letter,Place)
          In(Place)
     Eff: Add Delivrd(Letter,Place)
          Del Have(Letter) )
• Op(Action: Take(Letter,Place)
     Pre: Sender(Letter,Place)
          In(Place)
     Eff: Add Have(Letter) )
```

Planning Graphs

- A planner that uses a planning graph:
  A layered graph with arcs from one layer to the next.



- GraphPlan:
  - Searches in space of possible states
  - Searches for plans in two stages:
    1) **Expand**: From initial state to goal state(s)
    2) **Extract**: From goal state(s) back to initial state
  - Performs backward search in structured search space

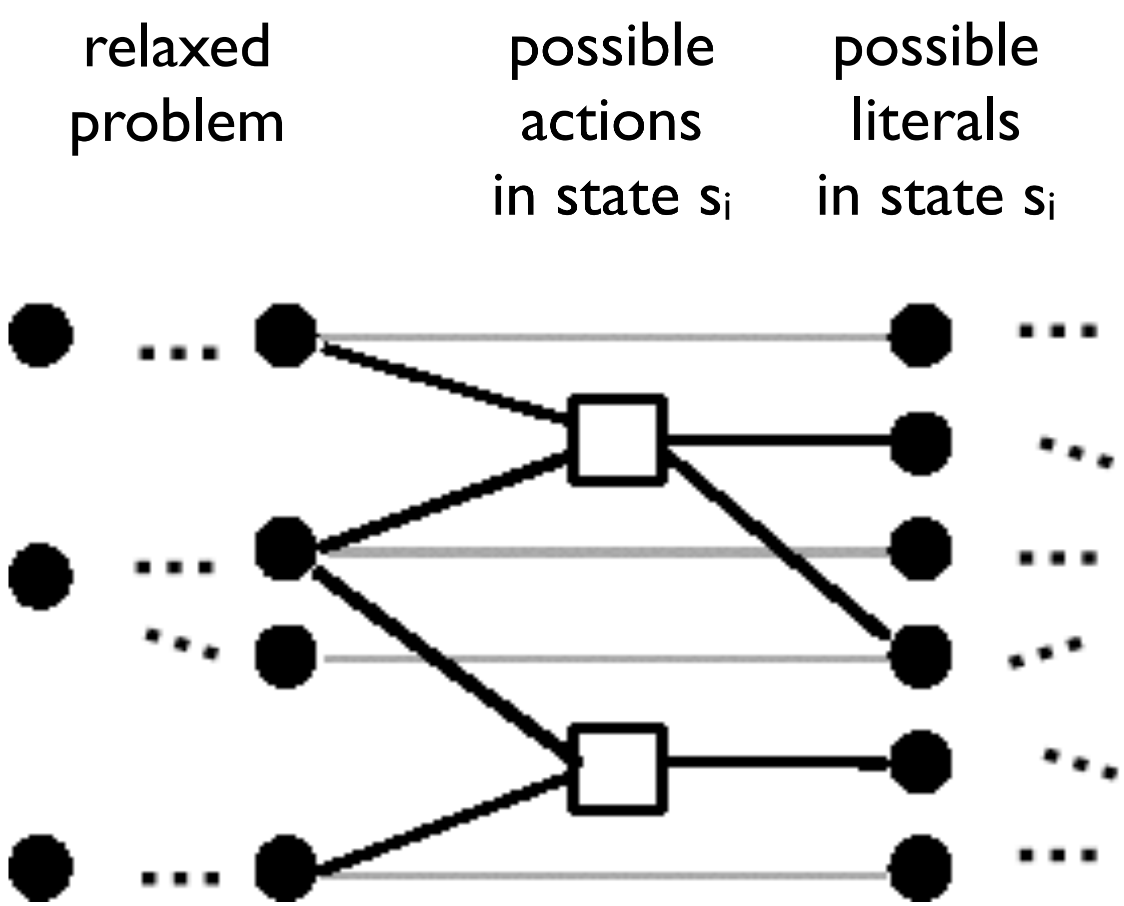
Planning Graphs

- Sequence of levels that correspond to time steps in the plan:
  - Level 0 consists only of
    - a single fact layer with all the literals of the initial state.
  - All other levels contain two layers:
    - An action layer, with all the actions whose preconditions are satisfied in the fact layer of the previous level
    - A fact layer with all the literals that produced by the effects of the actions

- Basic underlying idea:
  - Construct a superset of literals that could possibly be achieved by an n-level plan
  - Gives a compact representation of states that are reachable by n-level plans

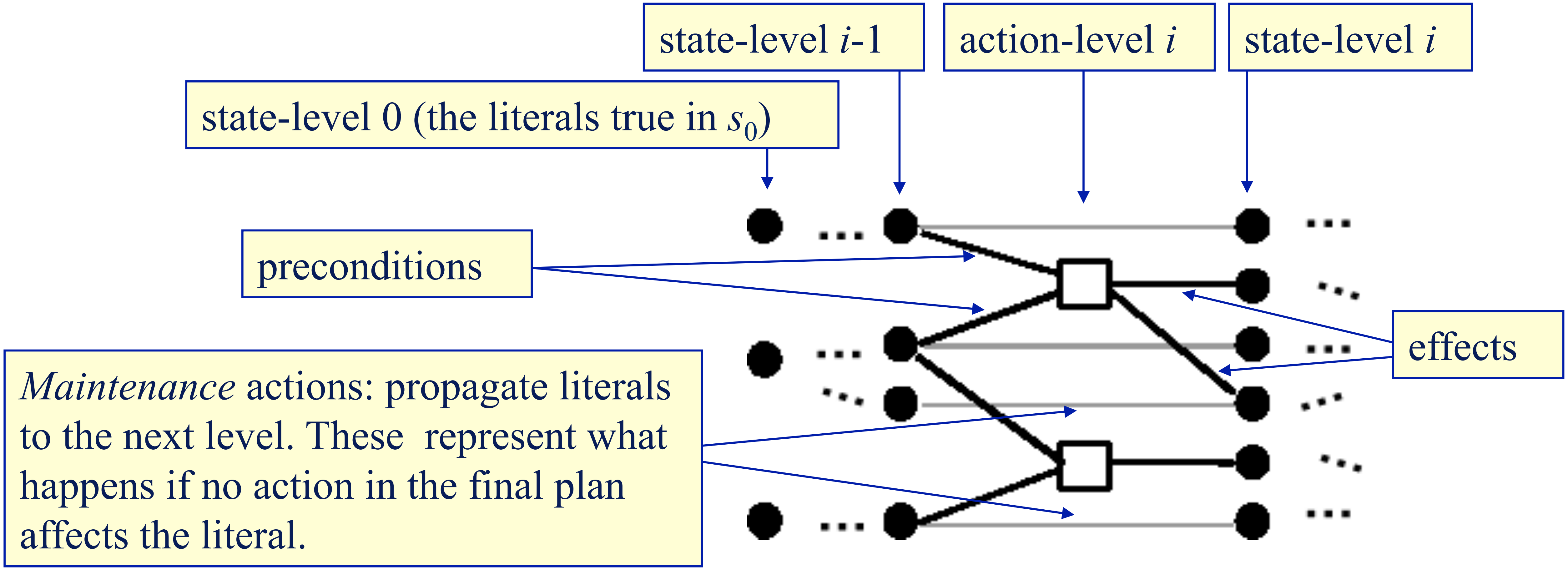
GraphPlan

- 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



The Planning Graph

- Alternating layers of ground literals and actions
  - All actions that might possibly occur at each time step
  - All of the literals asserted by those actions

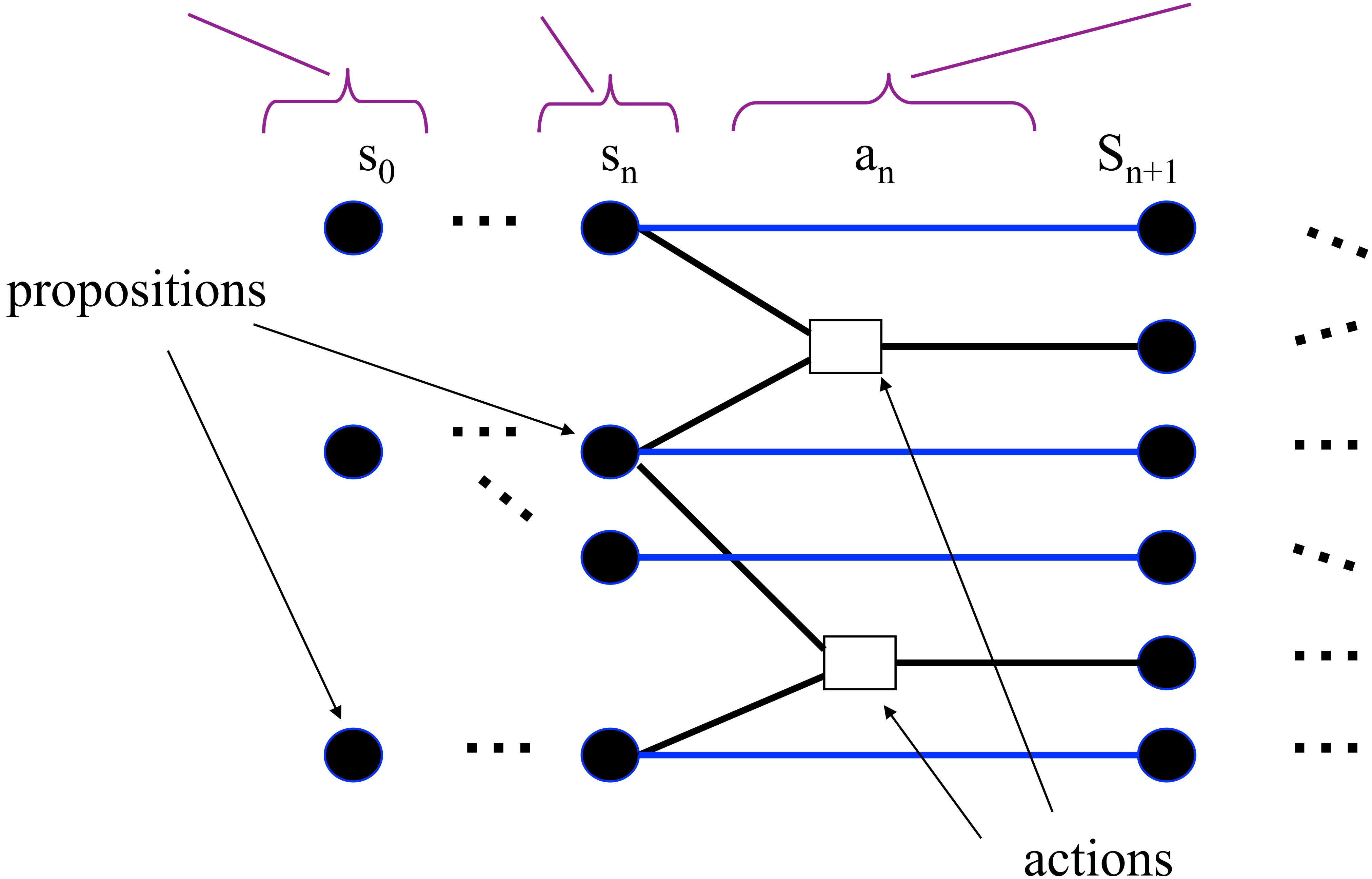


Planning Graphs: Levels

state-level 0: propositions true in s_0

state-level *n*: literals that may possibly be true after some *n* level plan

action-level *n*: actions that may possibly be applicable after some *n* level plan

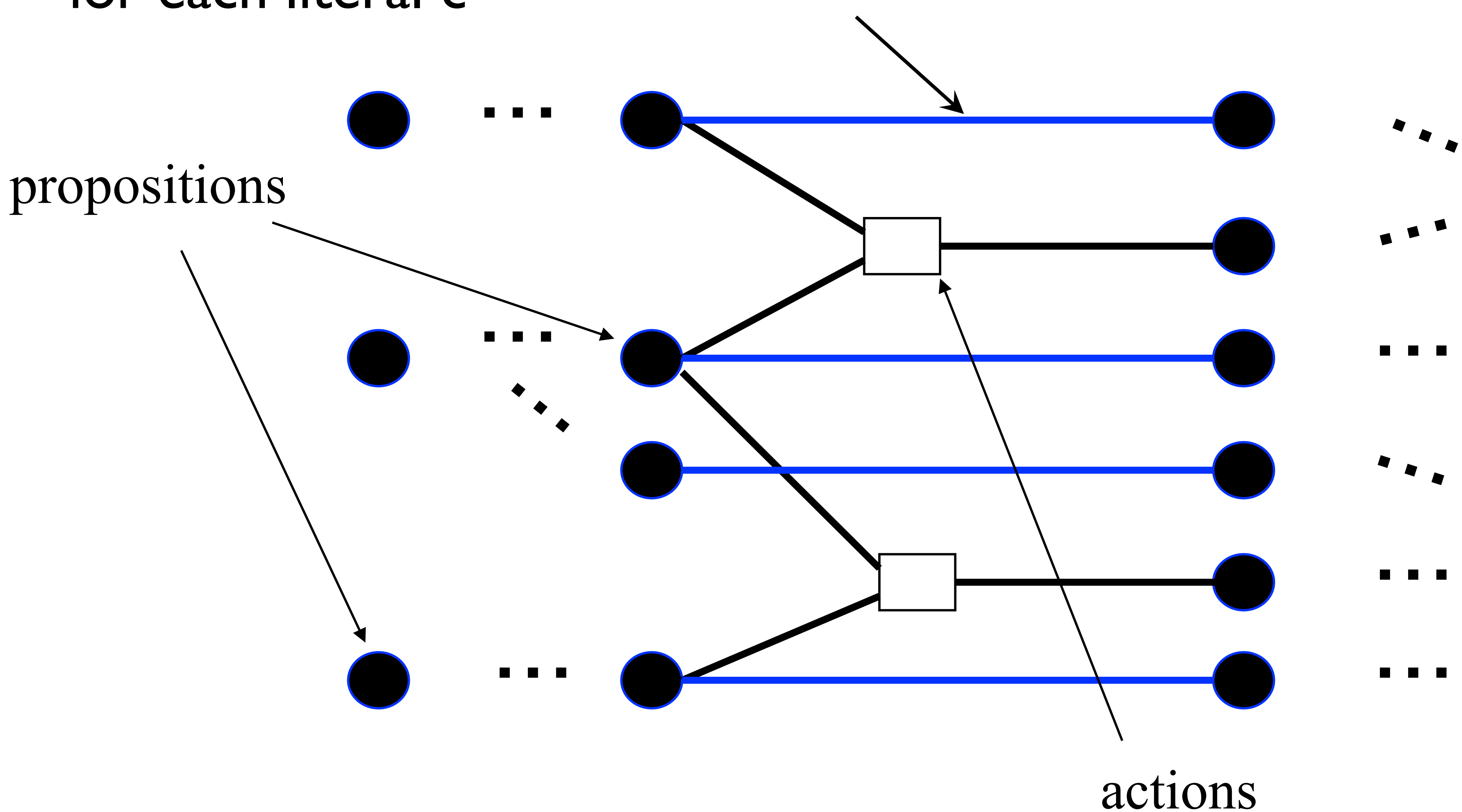


Planning Graphs: No-OPs

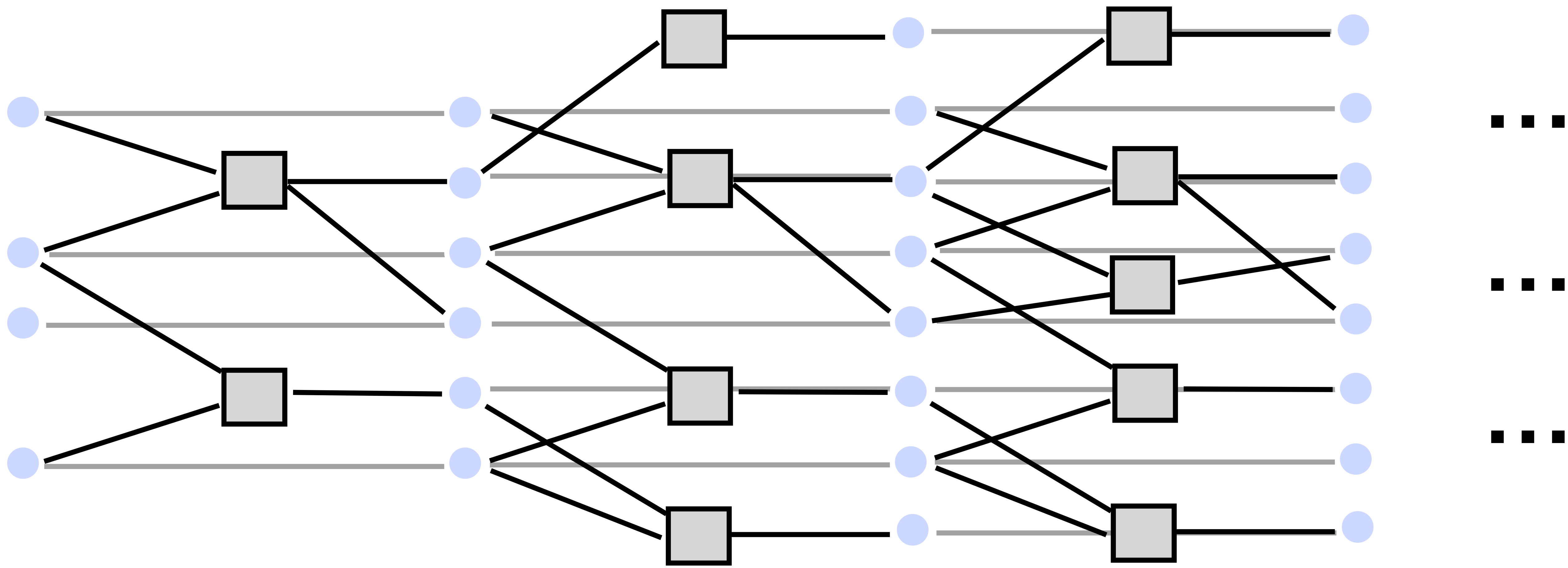
- The frame problem:
  - Which facts are still valid after execution of an action?
- Assumption: everything!
- We define special type of operators for every fact f of a fact level:
  - ```
    Op(Action: NOOP-F
         Pre: F
         Eff: F )
    ```
- NO-OPs copy all facts of a fact level onto the next fact level.
- NO-Ops are treated just like any other operator.
- Conclusion:
  A fact occurring once on a fact level
  will also occur on all succeeding fact levels.
    ```

Planning Graphs: No-OPs

- Maintenance action (persistence/no-op action)
  - Represents what happens if no action affects the literal
  - Equivalent to an action with precondition c and effect c, for each literal c



Planning graph



- **Definition**:
  An action is applicable on level Q_0
  iff its preconditions are contained in Q_0.

Stage 1: Graph Expansion

- Initial proposition layer
  - Just the initial state

- Action layer n
  - If all of an action's preconditions are in proposition layer n,
    (i.e. if it is applicable)
    then add action to layer n

- Proposition layer n+1
  - For each action at layer n (including persistence actions)
  - Add all its effects (both positive and negative) at layer n+1
    (Also allow propositions at layer n to persist to n+1: no-ops!)

- Propagate mutex information… (coming up)

Planning Graph

- **Definition 1**: A planning graph $\prod(N,E)$
  for a given planning problem $P(D,O,I,G)$
  is a directed bipartite graph with two types of nodes $N = A \cup F$
  - Action nodes $a \in A$
  - Fact nodes $f \in F$

  and two types of edges $E = PE \cup EE$
  - Precondition edges $pe \in PE \subseteq F \times A$
  - Effect edges $ee \in EE \subseteq A \times F$

  Every layer is made up of two levels:

  1) one fact level, consisting of literals only

  2) one action level, consisting of actions only

  The graph starts and ends with fact levels

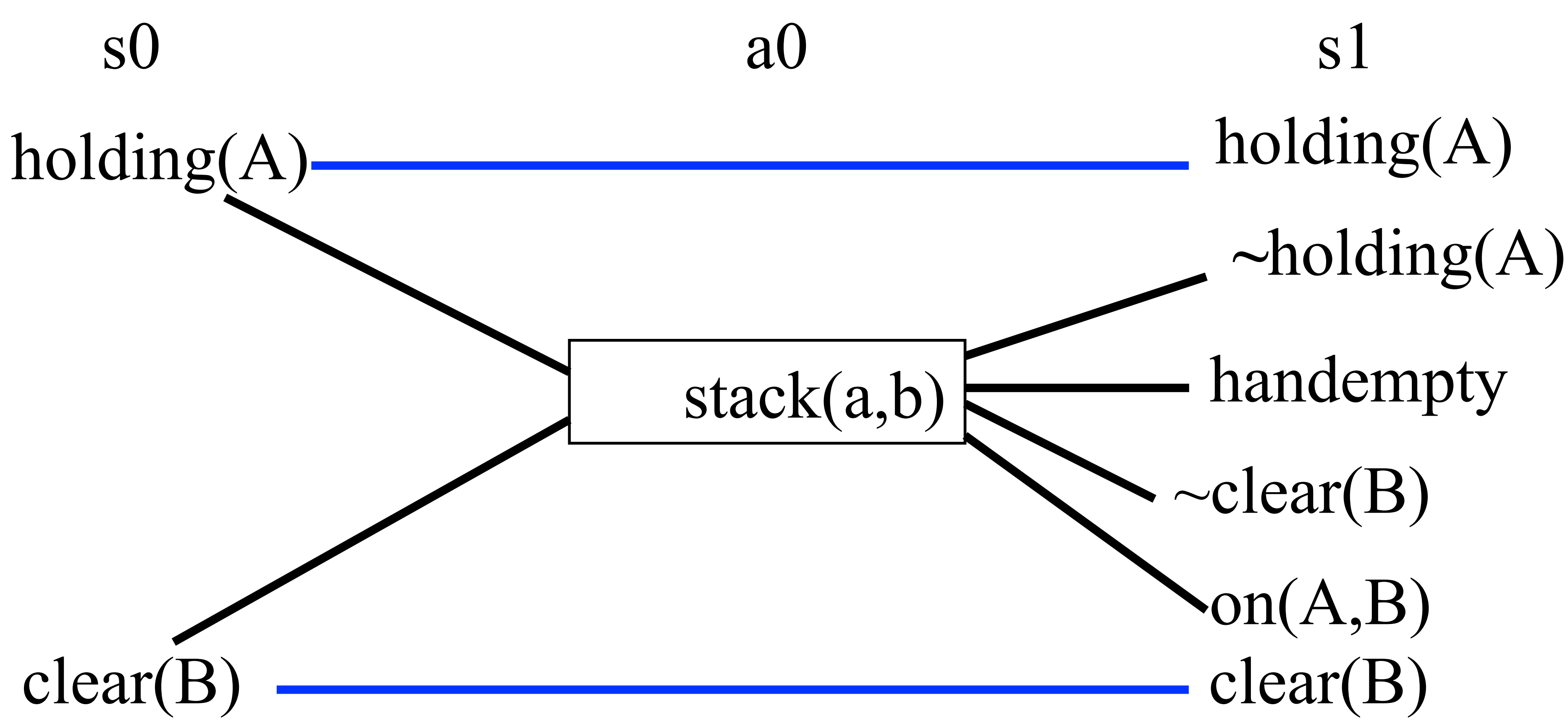
  Edges connect only nodes of neighboring levels

Dependencies of Plan Operators

- Parallel execution of actions
  - Actions can be executed in parallel as long as they do not interfere
    - Take(letter,office) & Deliver(parcel,office)
  - Agent is in same room for both actions;
    constraints on manipulation have not been formulated
    - Take(letter,office) & Deliver(parcel,lab)
  - This will cause failure!
    The agent must be in two different places at the same time.
- Thus, a formal definition of interference needed:
- **Definition 2**: Two actions interfere with each other,
  if one destroys an effect or a precondition of the other.
  - Such actions can never be executed in the same state
  - Plans with partially ordered interfering actions
    -- i.e. Which allow potential parallel execution of interfering actions --
    are never solutions to the planning problem
- This constitutes a very effective constraint for the search space

Example

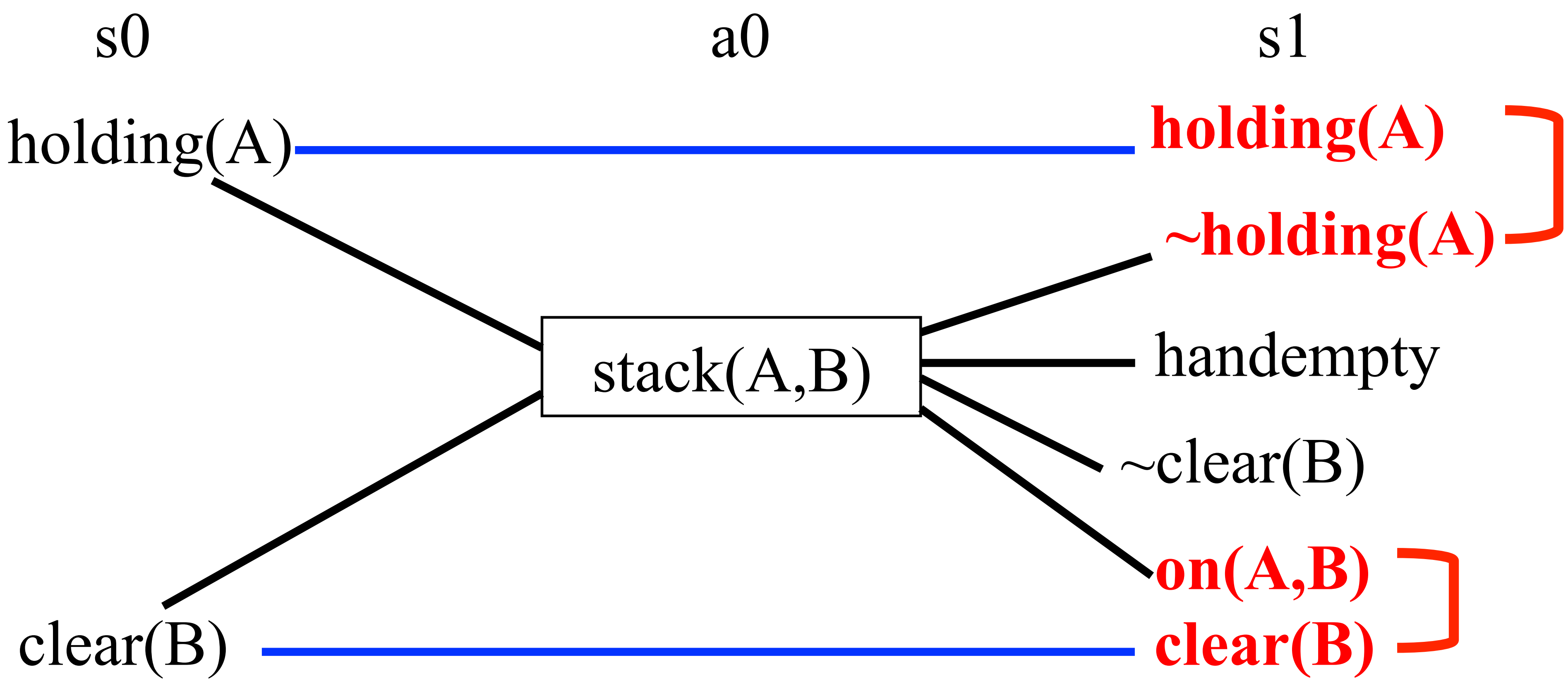
stack(A,B)
  precondition: holding(A), clear(B)
  effect: ~holding(A), ~clear(B), on(A,B), handempty



clear(B), holding(A) no-op actions…

Example

stack(A,B)
precondition: holding(A), clear(B)
effect: ~holding(A), ~clear(B), on(A,B), handempty



Notice that not all literals in s1 can be made true simultaneously after 1 level:
e.g. holding(A), ~holding(A) and on(A,B), clear(B)

Mutual Exclusion of Actions and Facts

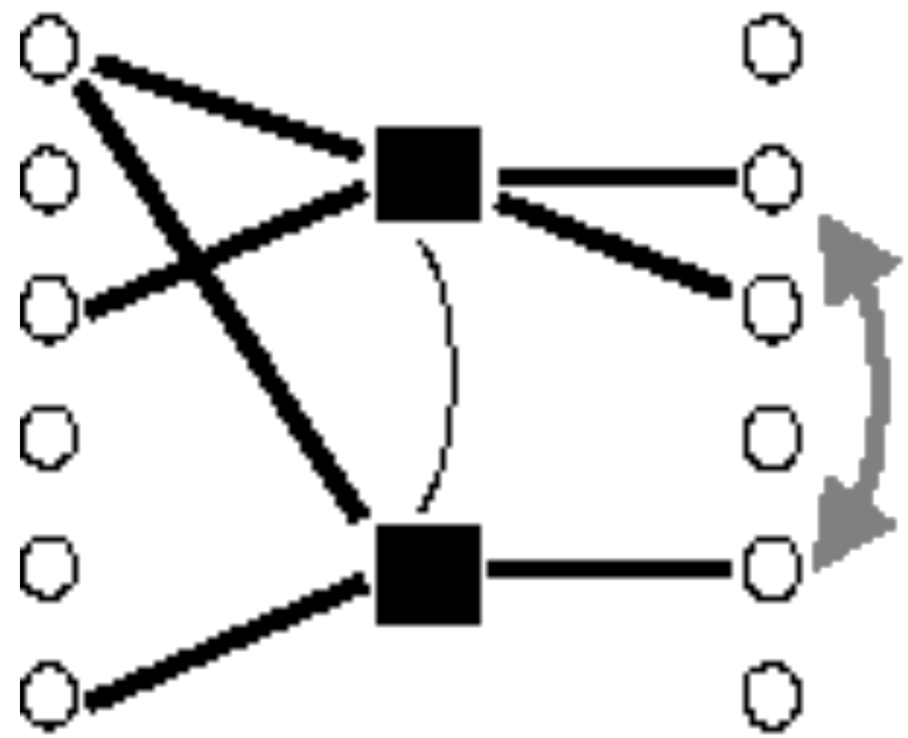
- **Definition 3**: An action is applicable on level Q_0 iff its preconditions are contained in Q_0.
- **Definition 4**: Two actions are mutually exclusive on level S_0 iff they interfere.
- **Definition 5**: Two facts p,q are mutually exclusive on level Q_i>=1
  iff there exists a pair of mutually-exclusive actions s,t on level S_{i-1}
  (provided mental health was present)
  (or no single action at all) which have p and q as effects.
- Any possible world state can never satisfy mutually exclusive facts!
- **Definition 6**: Two actions s,t have competing preconditions,
  iff there exist mutually exclusive facts p,q with $p \in pre(s) \quad q \in pre(t)$
- **Definition 7**: Two actions s,t, are mutually exclusive
  - On level S_0 iff they interfere
  - On level $S_{i>=1}$ iff they interfere or have competing preconditions
- In any possible world state, mutually exclusive actions are never concurrently executable.

Mutual Exclusion (Mutex)

- Between pairs of actions
  - No valid plan could contain both at layer n
  - E.g., stack(a,b), unstack(a,b)
- Between pairs of literals
  - No valid plan could produce both at layer n
  - E.g., clear(a), ~clear(a)
    on(a,b), clear(b)
- GraphPlan checks pairs only
  - Mutex relationships help rule out possibilities during search in Stage 2 of GraphPlan

Action Mutex 1: Inconsistent Effects

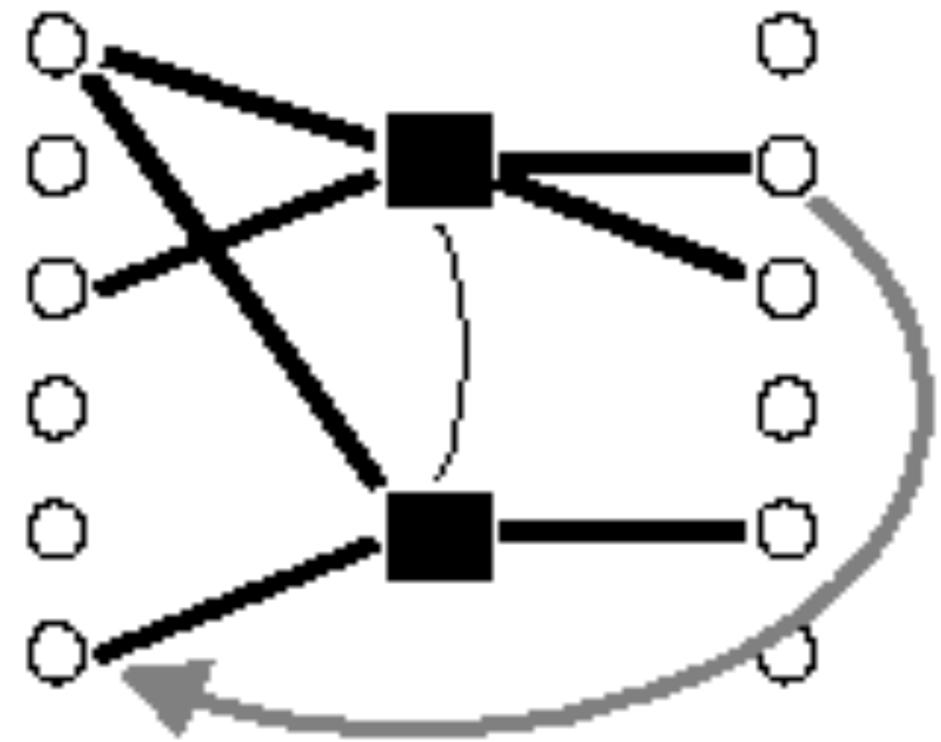
- Inconsistent effects
  - An effect of one action negates an effect of another action
- Example:
  - stack(a,b) & unstack(a,b)
  - adds handempty deletes handempty



Inconsistent Effects

Action Mutex 2: Interference

- Interference:
  - One action deletes a precondition of another action



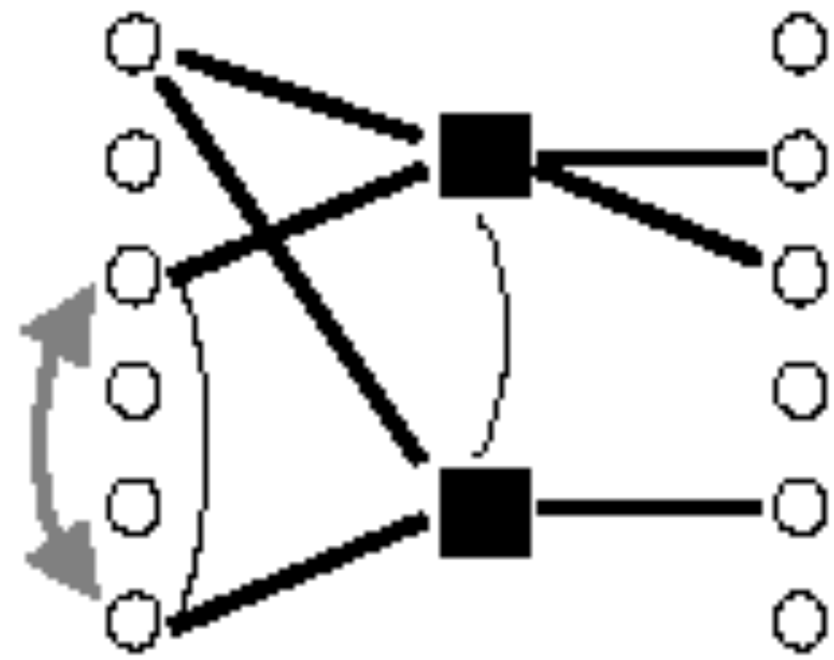
Interference

- Example:
  - stack(a,b) & putdown(a)
  - deletes holding(a) needs holding(a)

- **Definition 4**: Two actions are mutually exclusive on level S_0 iff they interfere

Action Mutex 3: Competing Needs

- Competing needs:
  - The actions have mutually exclusive preconditions
  - It follows that these preconditions cannot be true at the same time

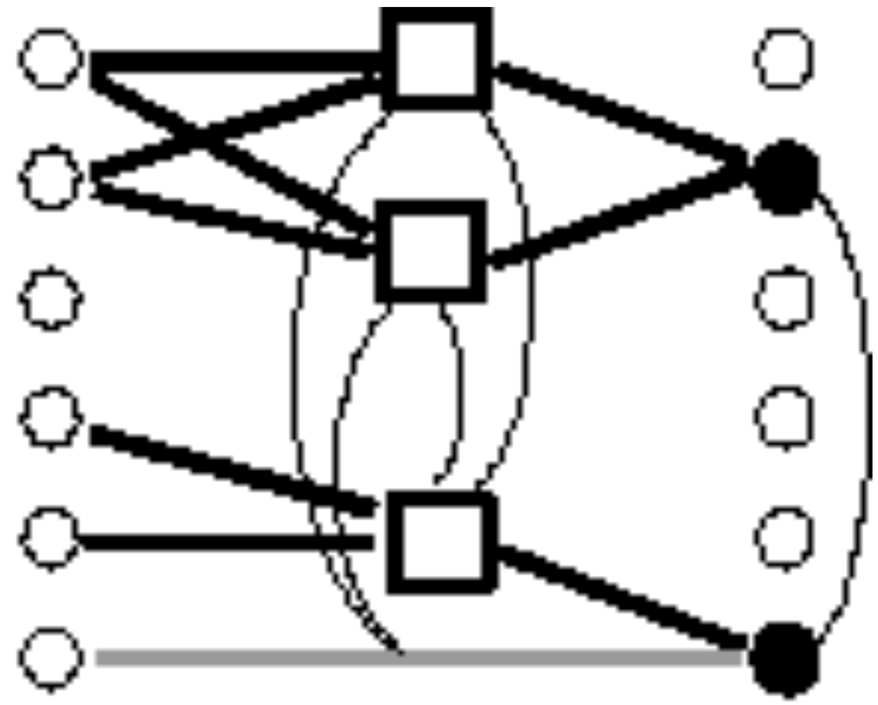


Competing Needs

- **Definition 6**: Two actions s,t have competing preconditions, iff there exist mutually exclusive facts p,q with $p \in pre(s)$ $q \in pre(t)$

Literal Mutex: Inconsistent Support

- Inconsistent support :
  - One is the negation of the other
    E.g., handempty and ~handempty
  - All ways of achieving them via actions are pairwise mutex



Inconsistent Support

- **Definition 5**: Two facts p,q are mutually exclusive on level $Q_{i>=1}$ iff there exists a pair of mutually-exclusive actions s,t on level S_{i-1} (or no single action at all) which have p and q as effects.
- Any possible world state can never satisfy mutually exclusive facts!

Example

- (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

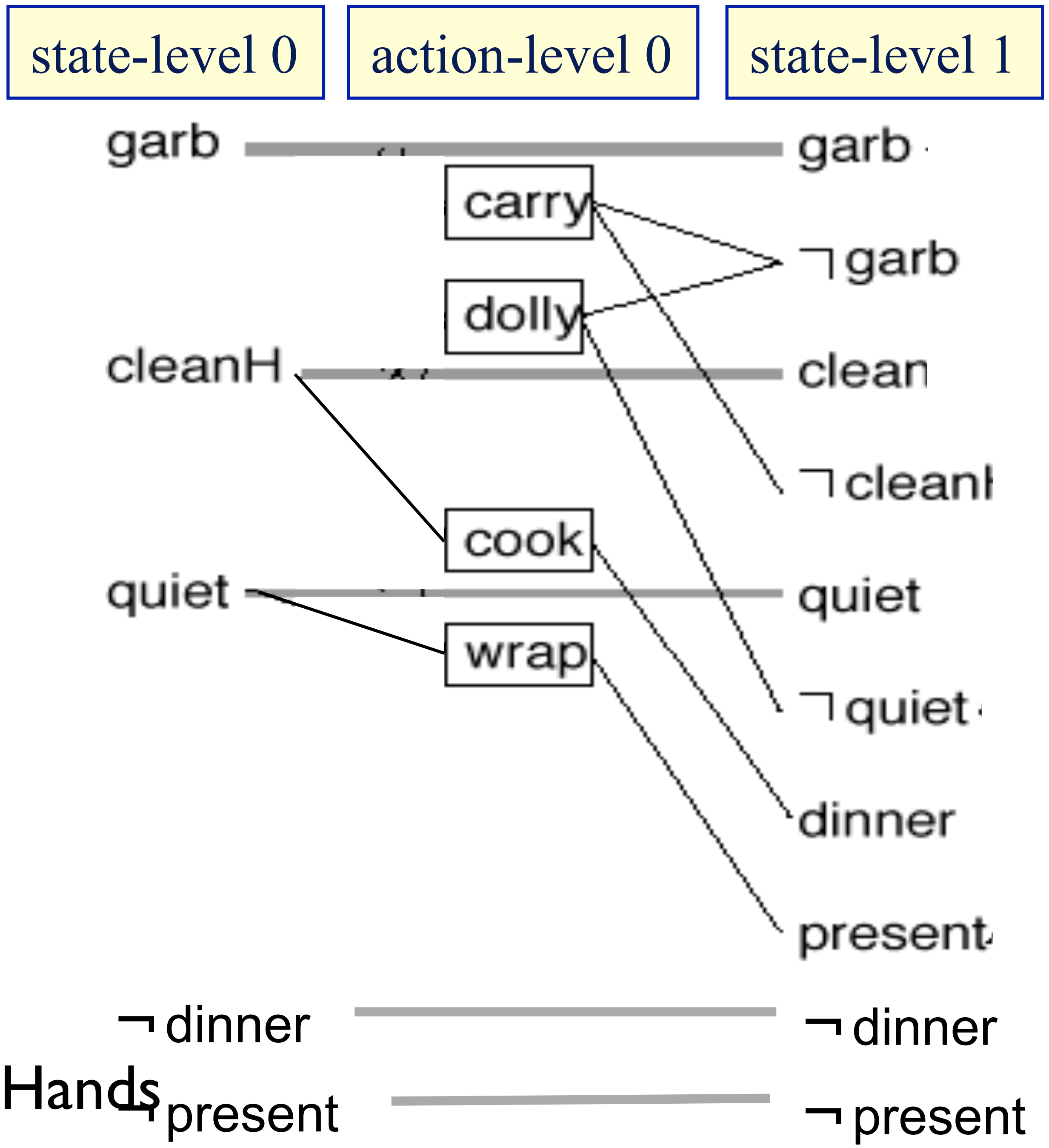
- Also have maintenance actions: one for each literal

Example (continued)

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

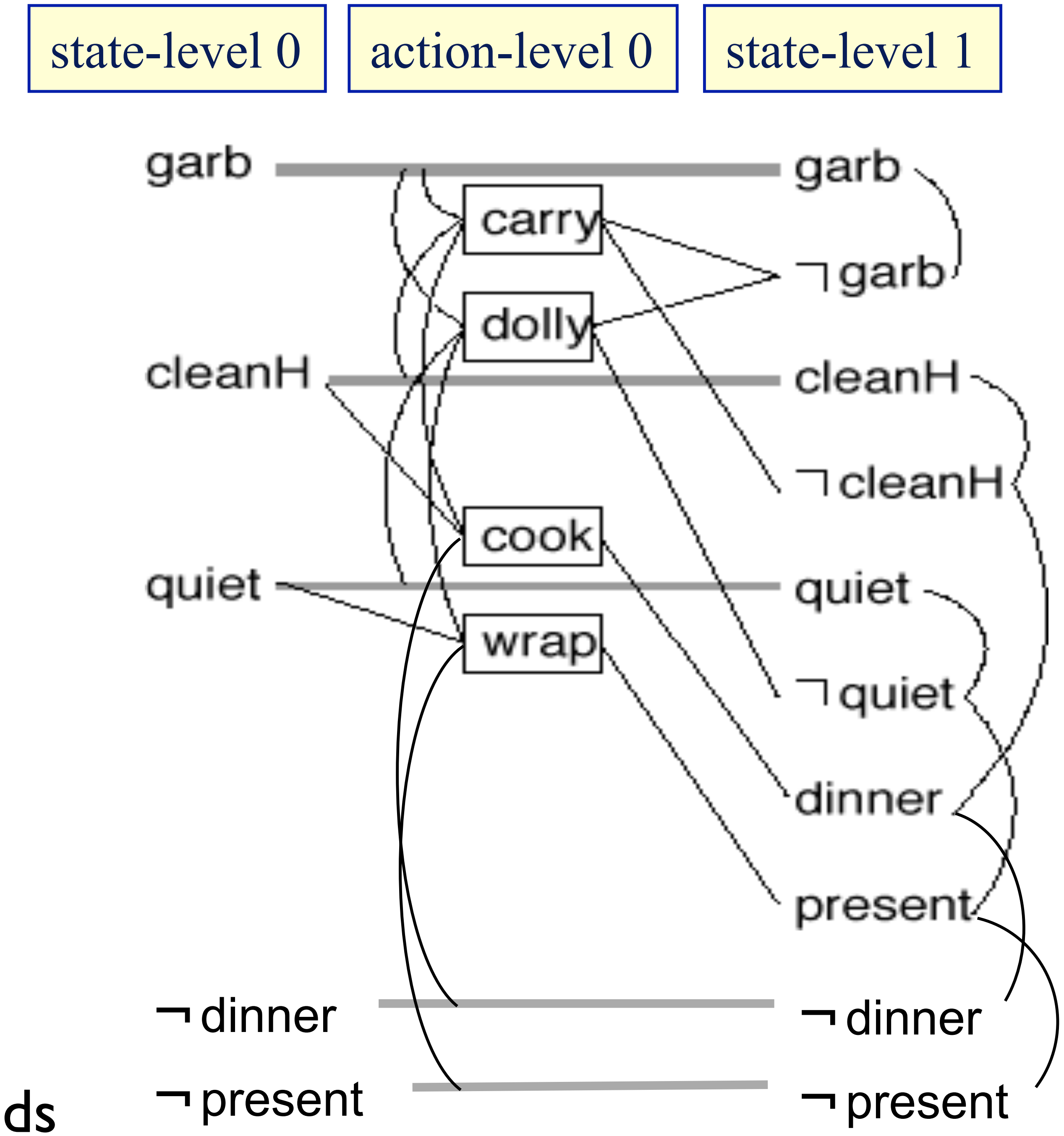
Action	Preconditions	Effects
cook()	cleanHands	dinner
wrap()	quiet	present
carry()	none	¬garbage, ¬cleanHands
dolly()	none	¬garbage, ¬quiet

- Also have the maintenance actions



Example (continued) - Check for mutexes

- 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 action
- Action   Preconditions   Effects
- cook()   cleanHands   dinner
- wrap()   quiet   present
- carry()   none   ¬garbage, ¬cleanHands
- dolly()   none   ¬garbage, ¬quiet
- Also have the maintenance actions



Stage 2: Plan Extraction by Backward Search

- As soon as latest fact level contains all sub-goals, a plan can possibly be extracted.
- Start with maximal (latest) fact level
- Choice point:
  - Minimal set of non-exclusive operators on level i which achieve goal on level i+1
  - Minimal means that each operator achieves at least one sub-goal which is not produced by any other operator
- Preconditions of the selected actions define goals on level i
- Backtrack, if any pair of actions or goals are mutually exclusive.

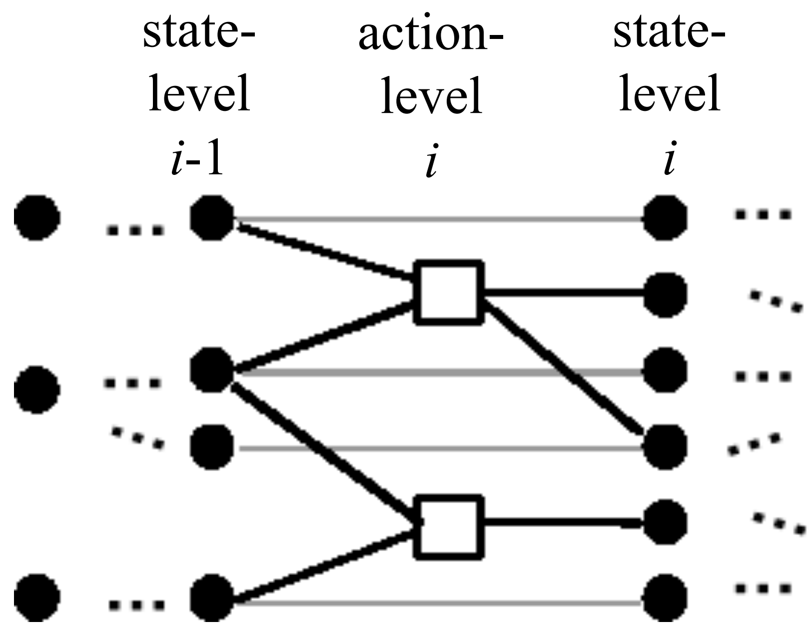
Solution Extraction

```
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
```

The level of the state s_j

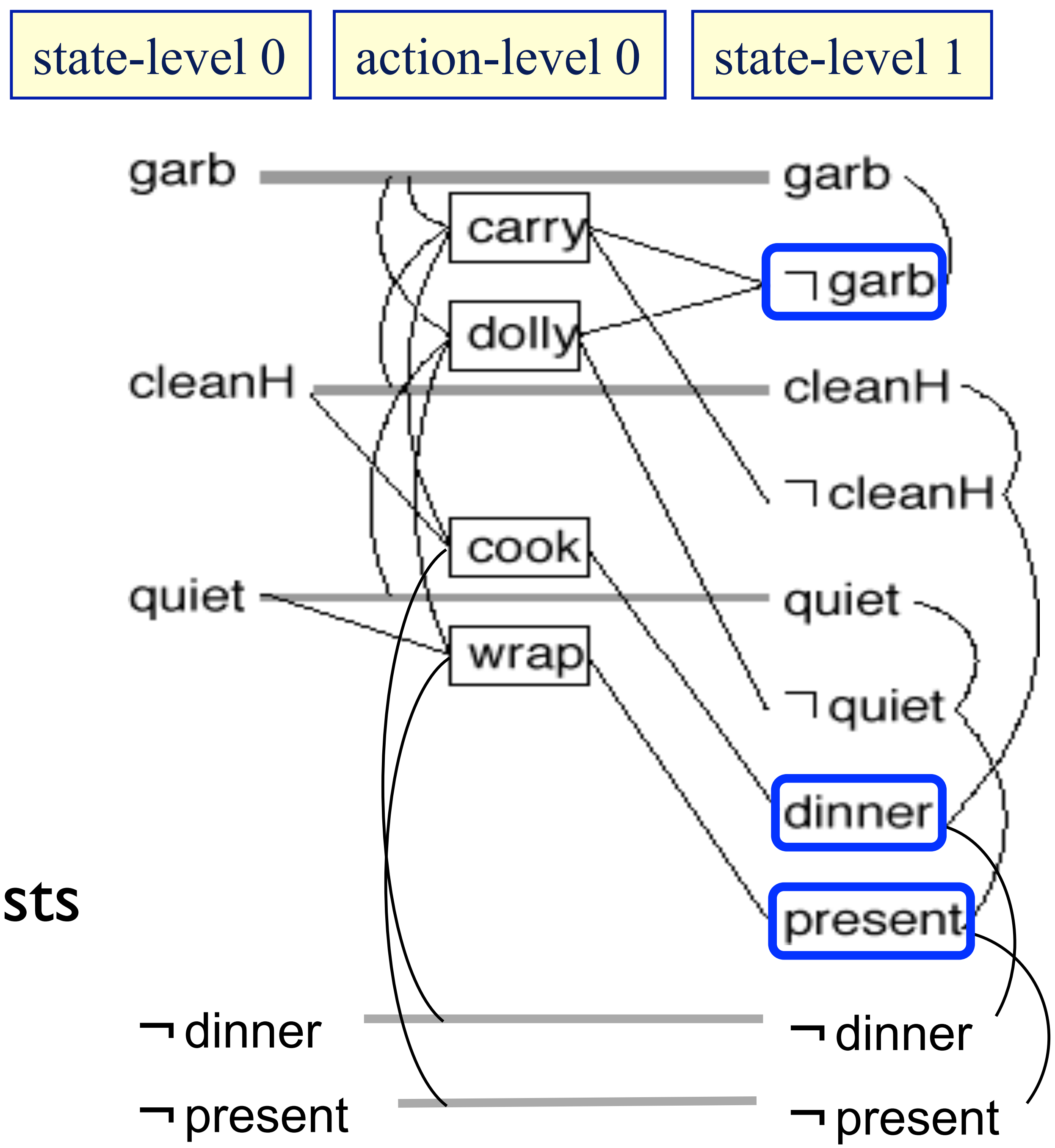
The set of goals we are trying to achieve

A real action or a maintenance action



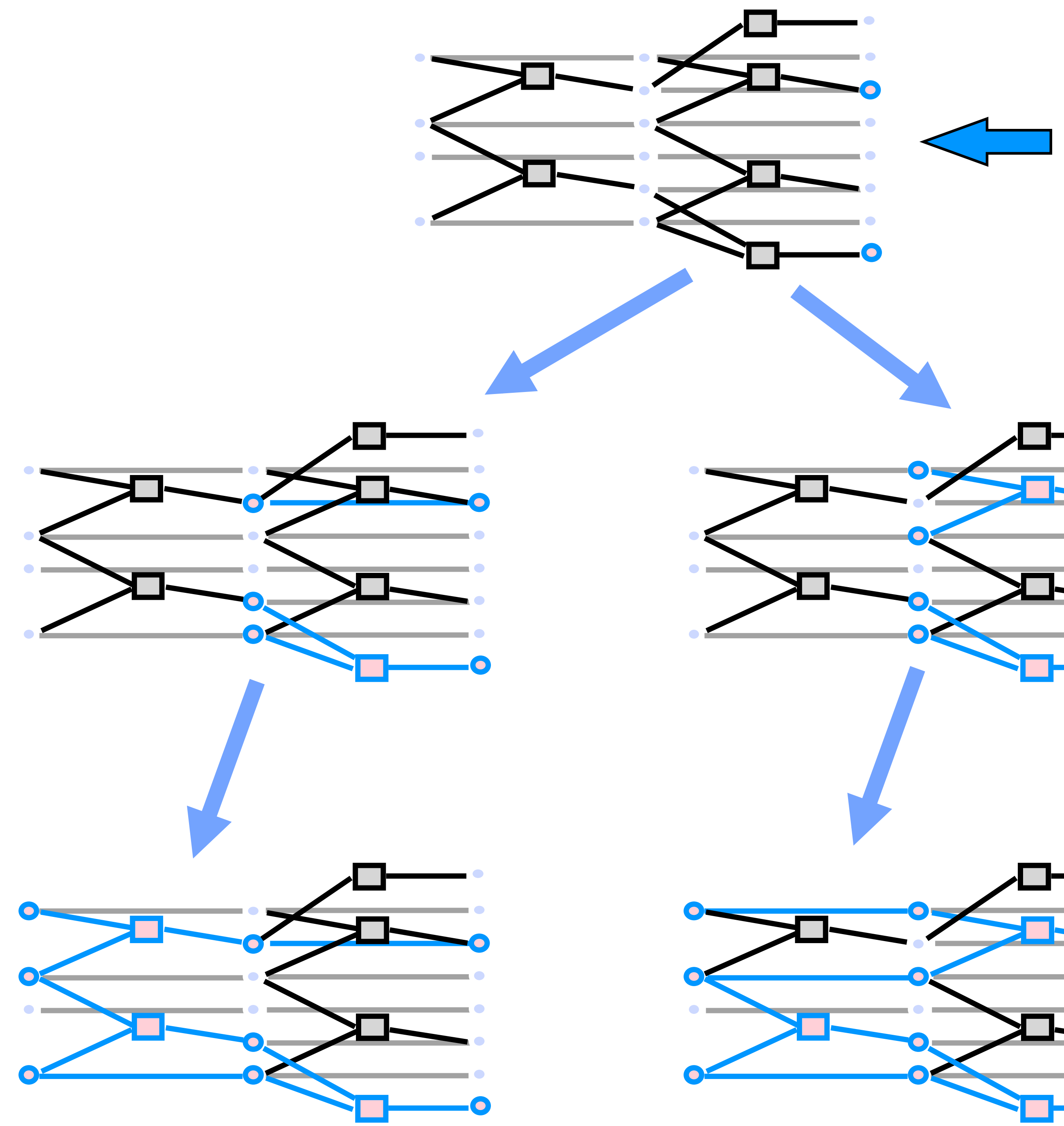
Example (continued) - Do we have a solution?

- Check to see whether there is a possible plan
- Recall that the goal is
  - {¬garbage, dinner, present}
- Note that
  - All are possible in s_1
  - None are mutex with each other
- Thus, there's a chance that a plan exists
- Try to find it
  - Solution extraction



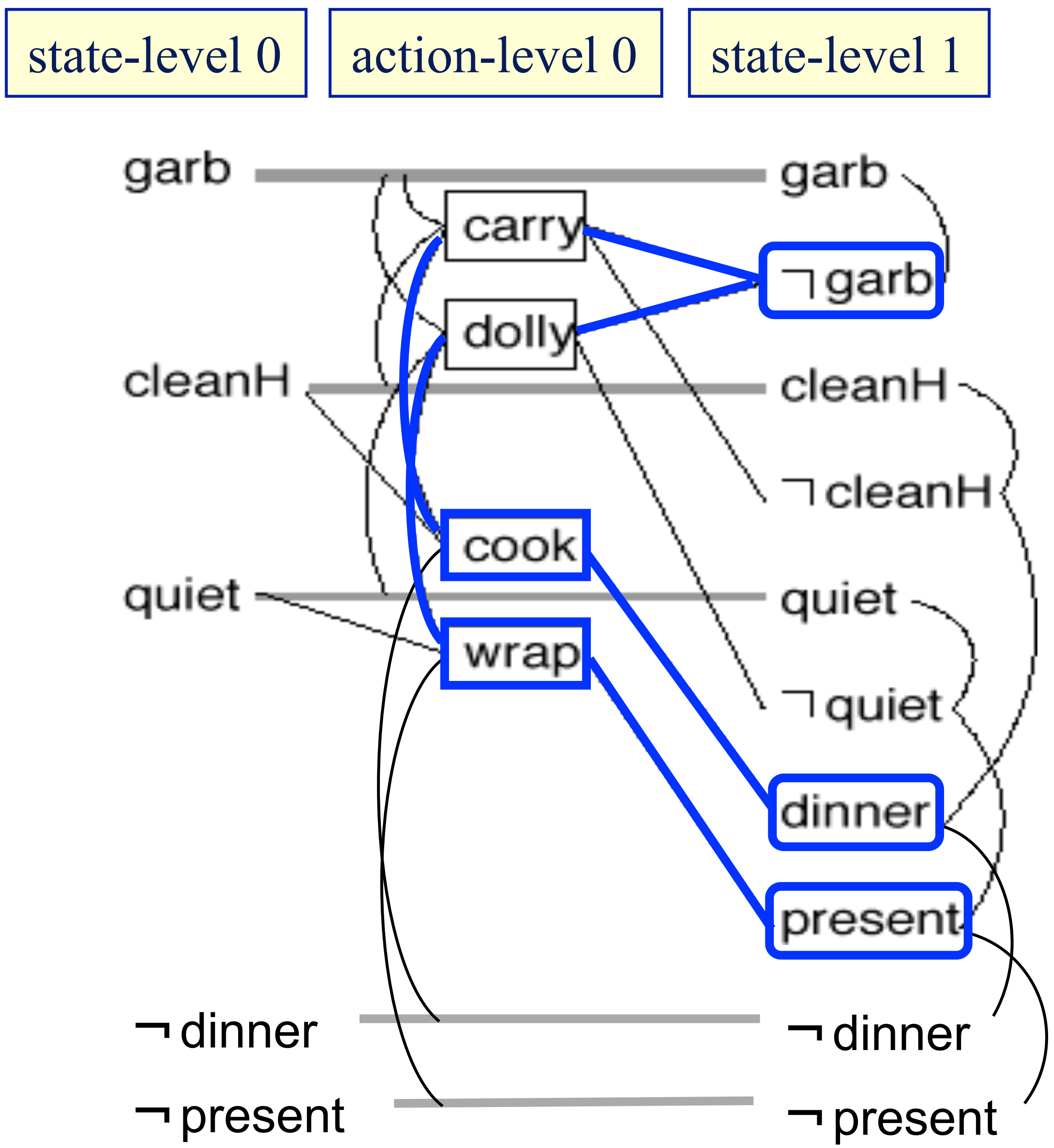
Planning Graph: Solution Extraction Search

- If goals are present & non-mutex:
  - Choose action to achieve each goal
  - Add preconditions to next goal set



Example (continued) - Possible solutions

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

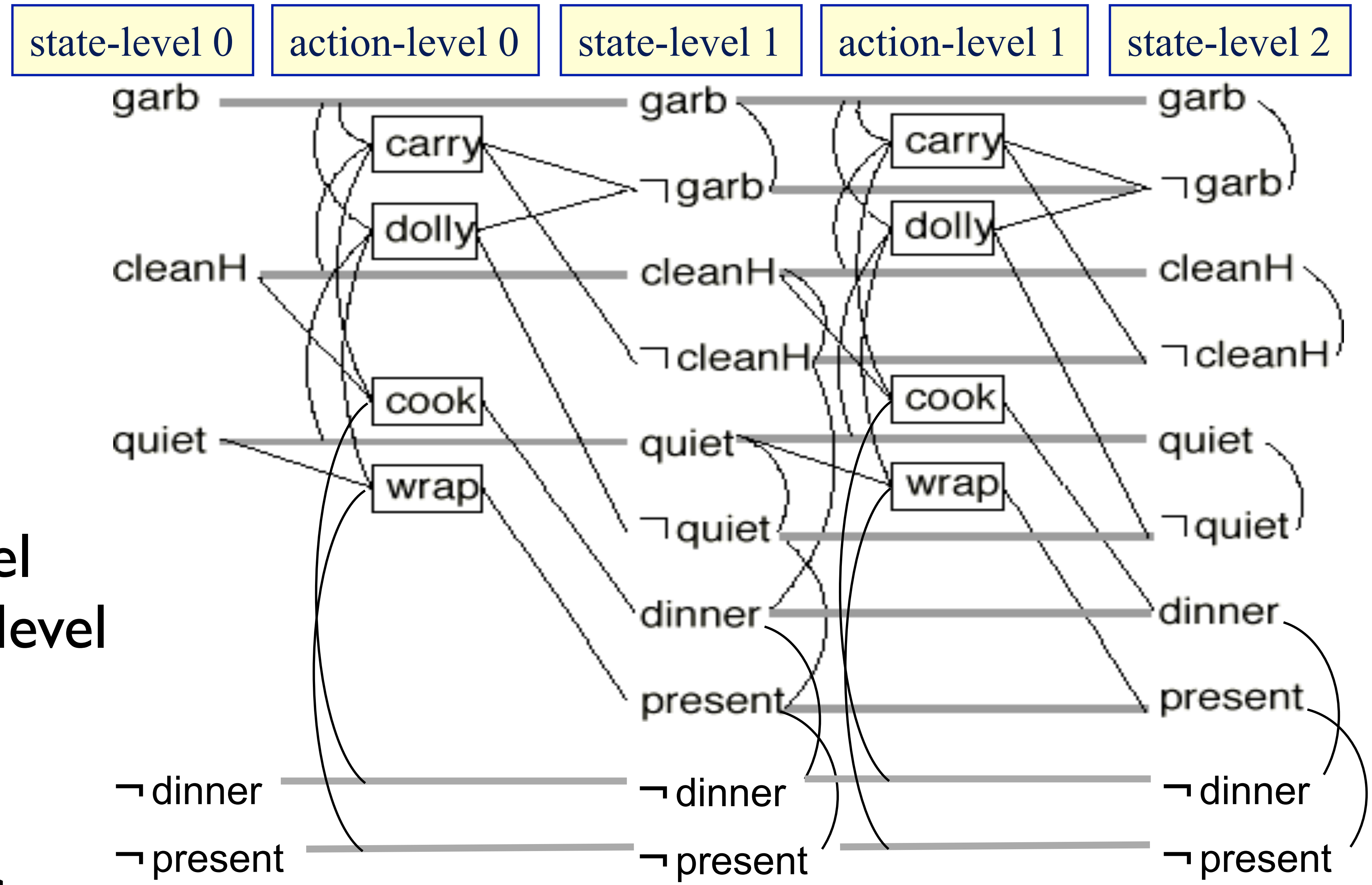


Example (continued): Add a new layer

- Backtrack
  and do more
  graph expansion

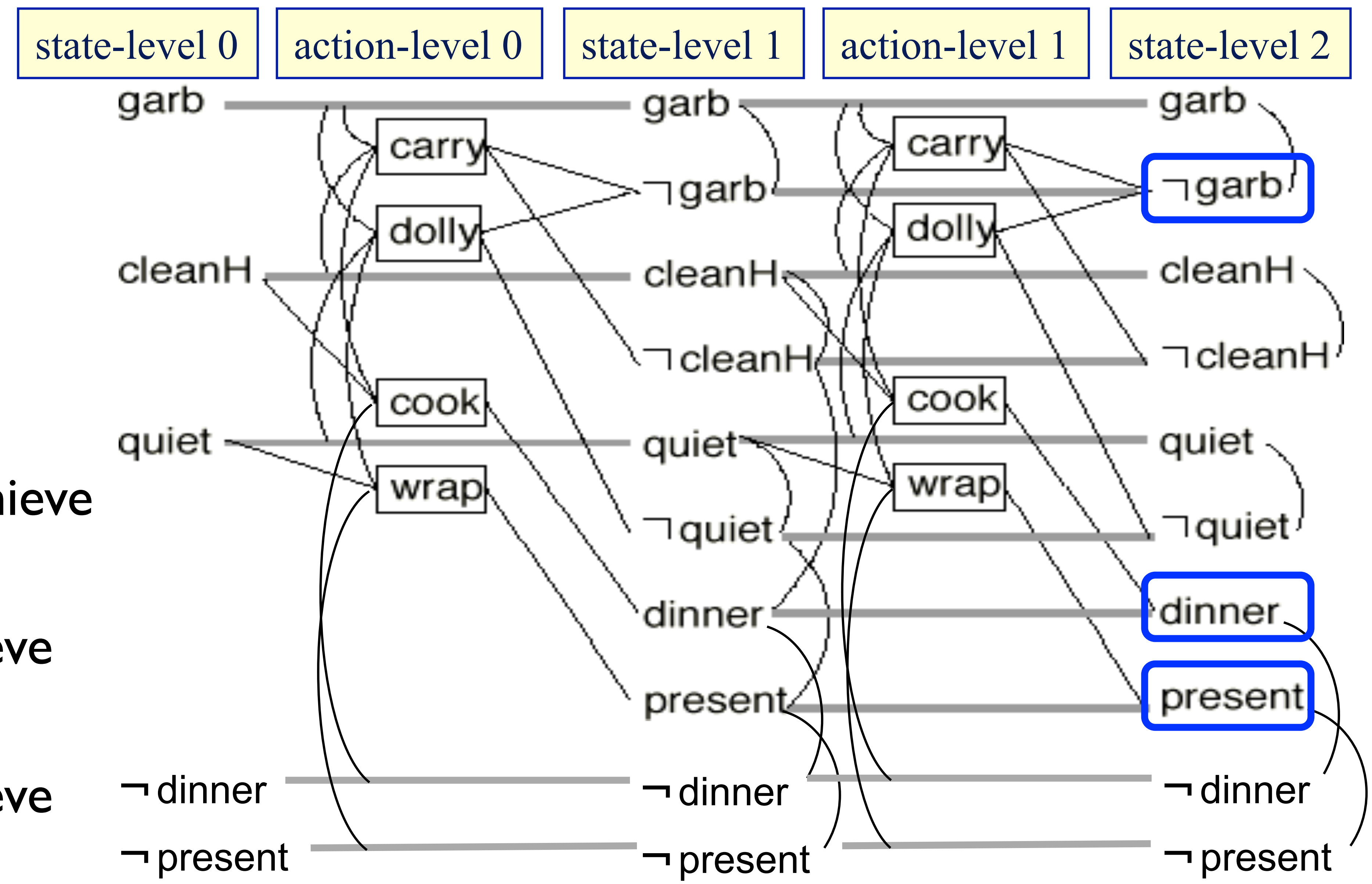
- Generate
  another action level
  and another state level

- Adding a layer
  provides new ways
  to achieve propositions!



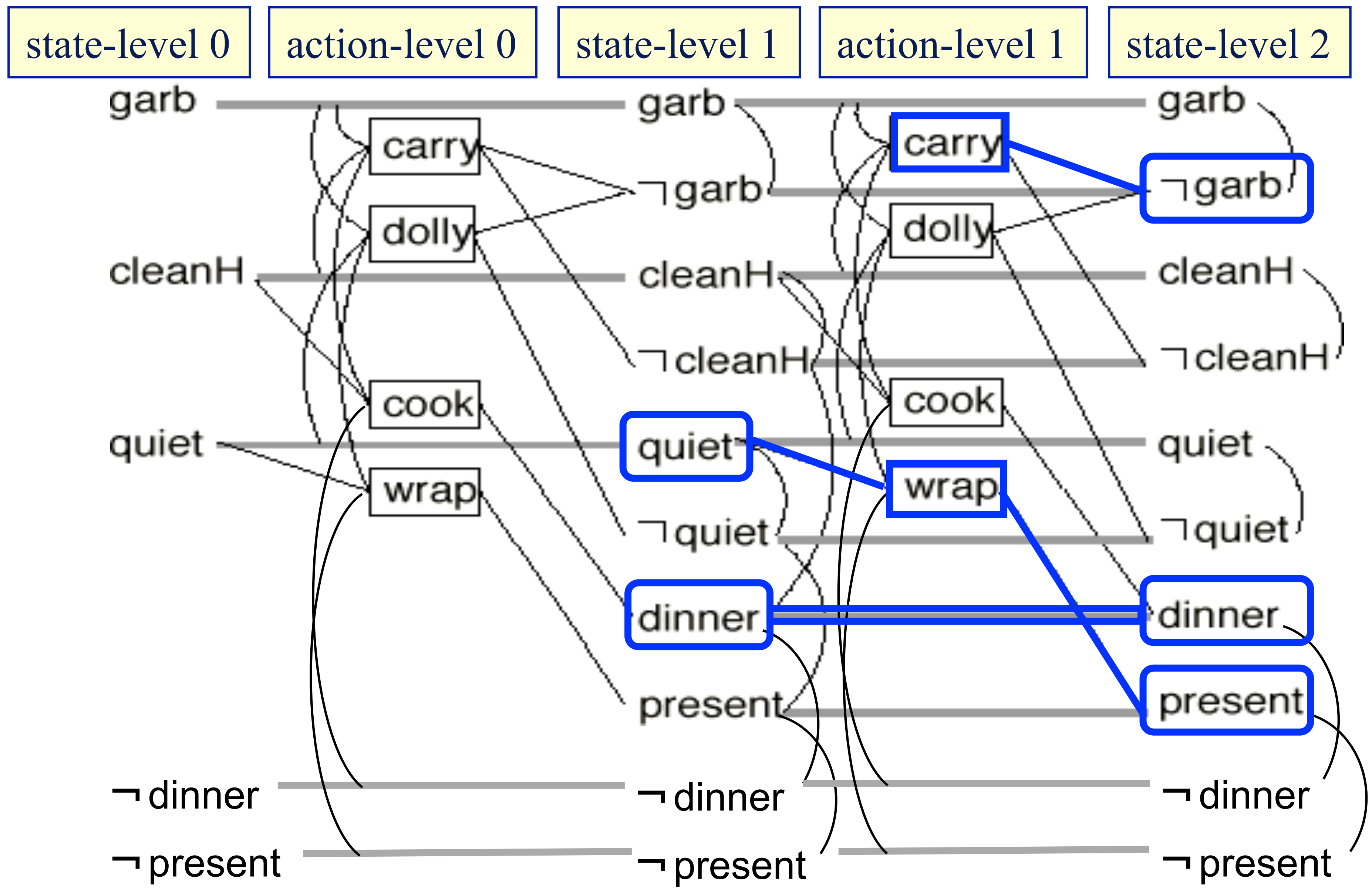
Example (continued) - Now, do we have a solution?

- Solution extraction
- Twelve combinations at action level 1
  - Three ways to achieve ¬garb
  - Two ways to achieve dinner
  - Two ways to achieve present



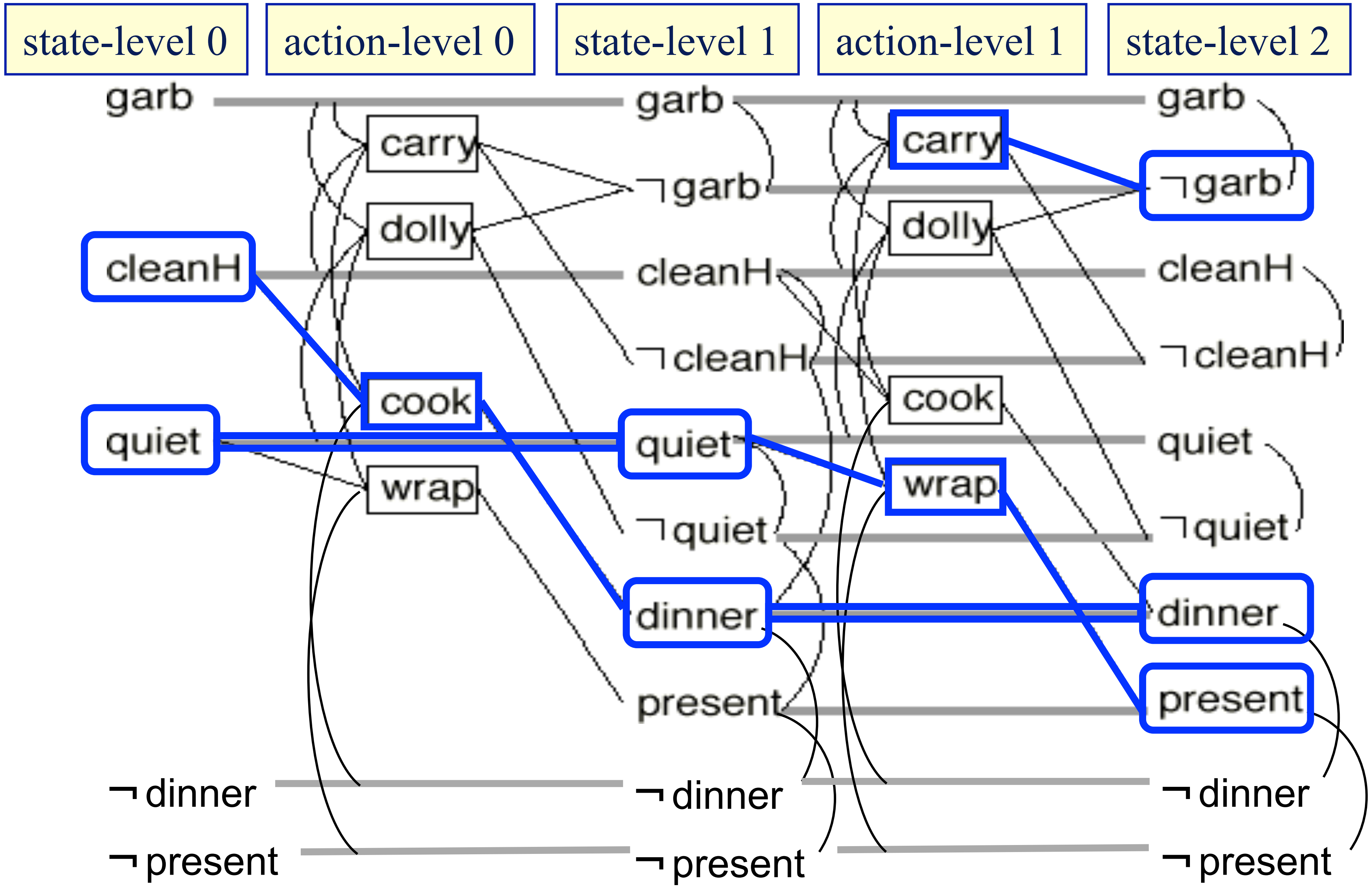
Example (continued) - Plan extraction

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



Example (continued) - Success!

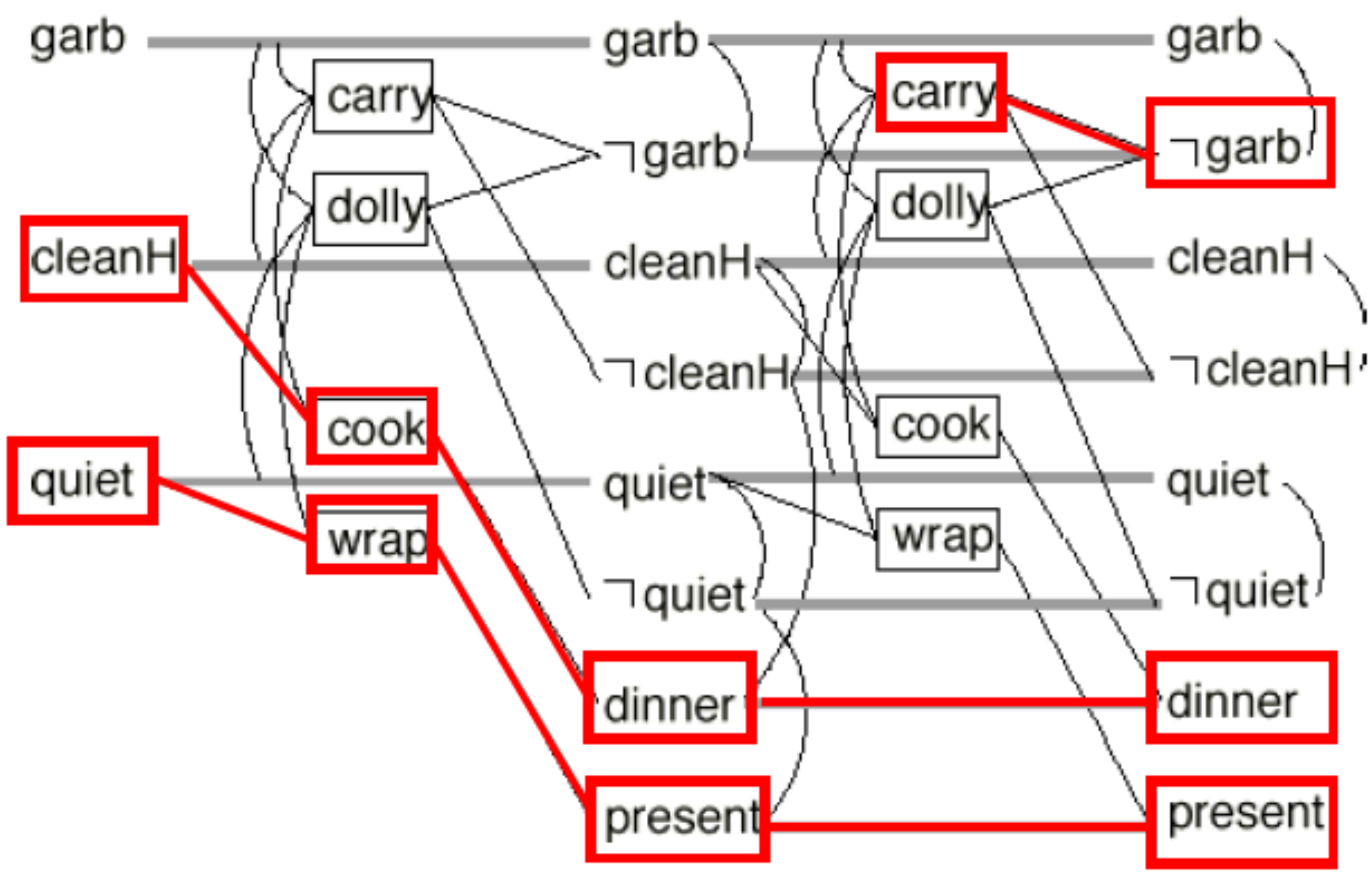
- Call Solution-Extraction recursively at level 1
- The action set {cook, noop-quiet} at layer 1 supports preconditions. Their preconditions are satisfied in initial state, so we have found a solution!



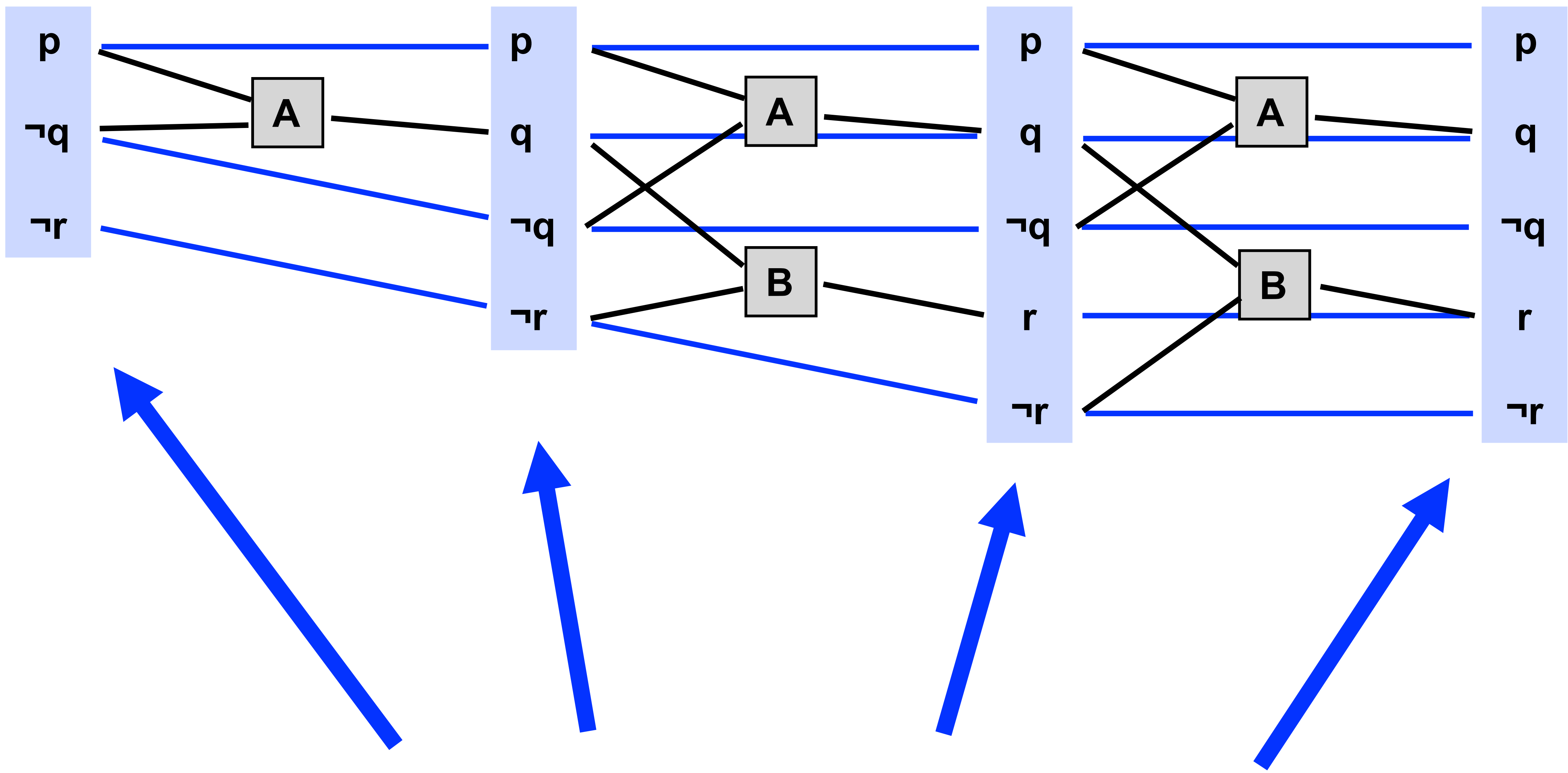
- Solution whose parallel length is 2: <{cook},{carry, wrap}>

Example (continued) - Another solution

- Another solution {cook,wrap} ; {carry}

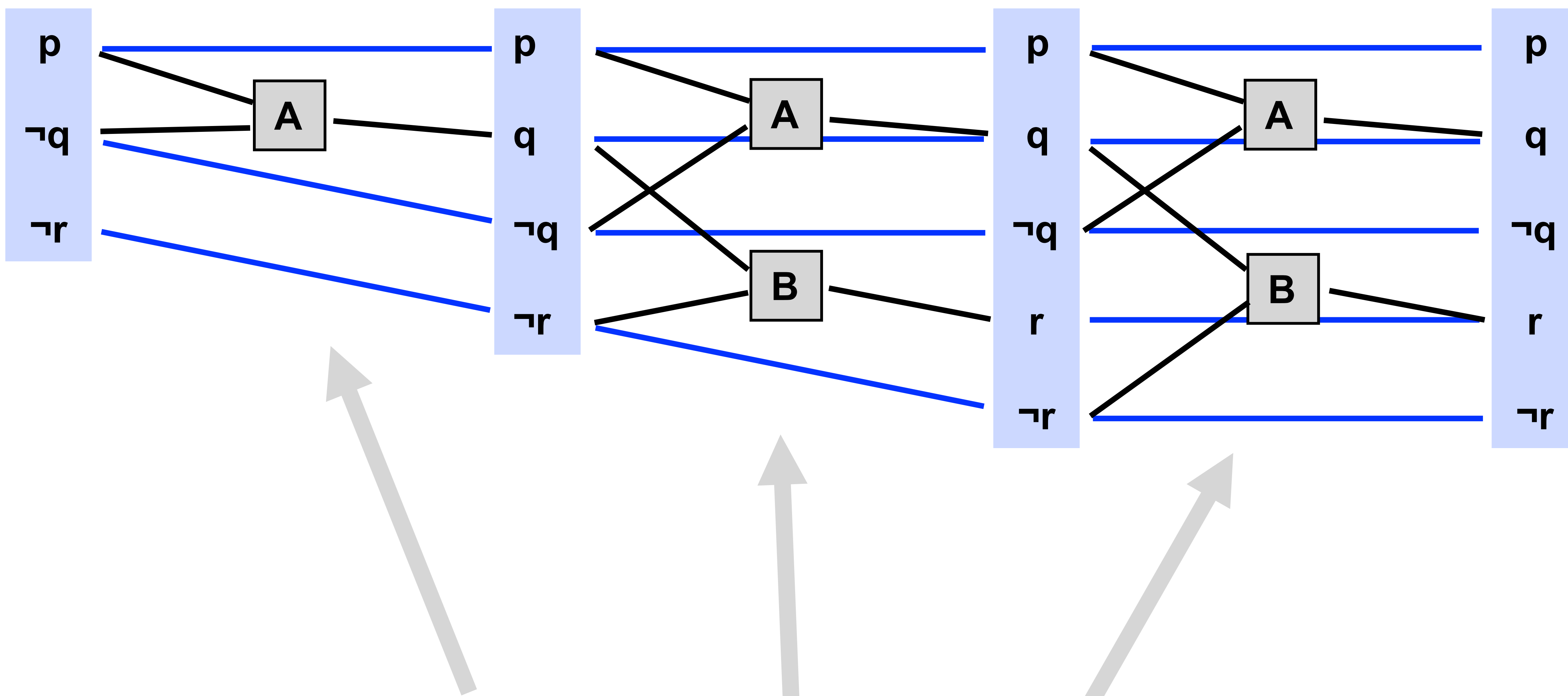


Property 1



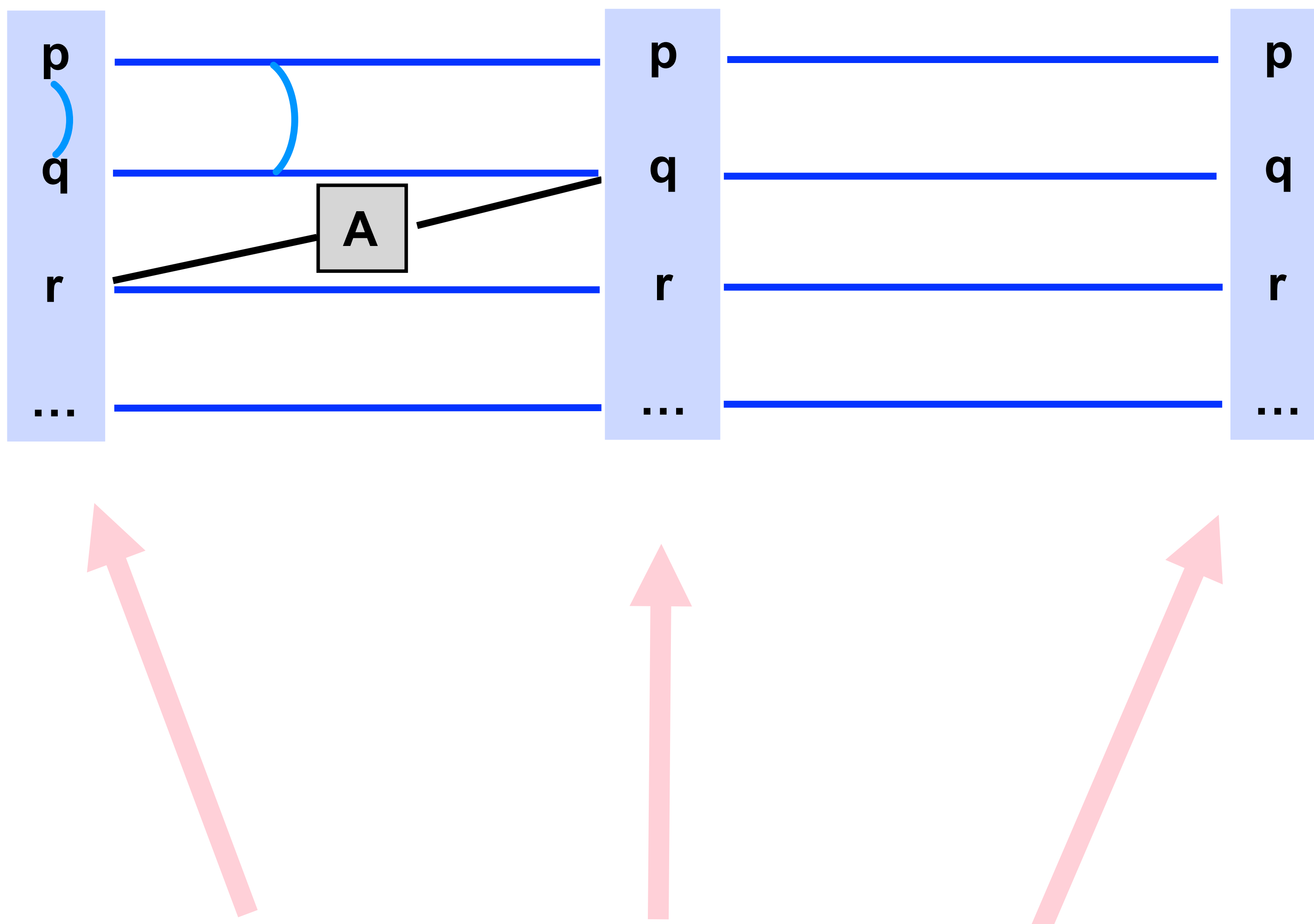
Propositions monotonically increase
(always carried forward by no-ops)

Property 2



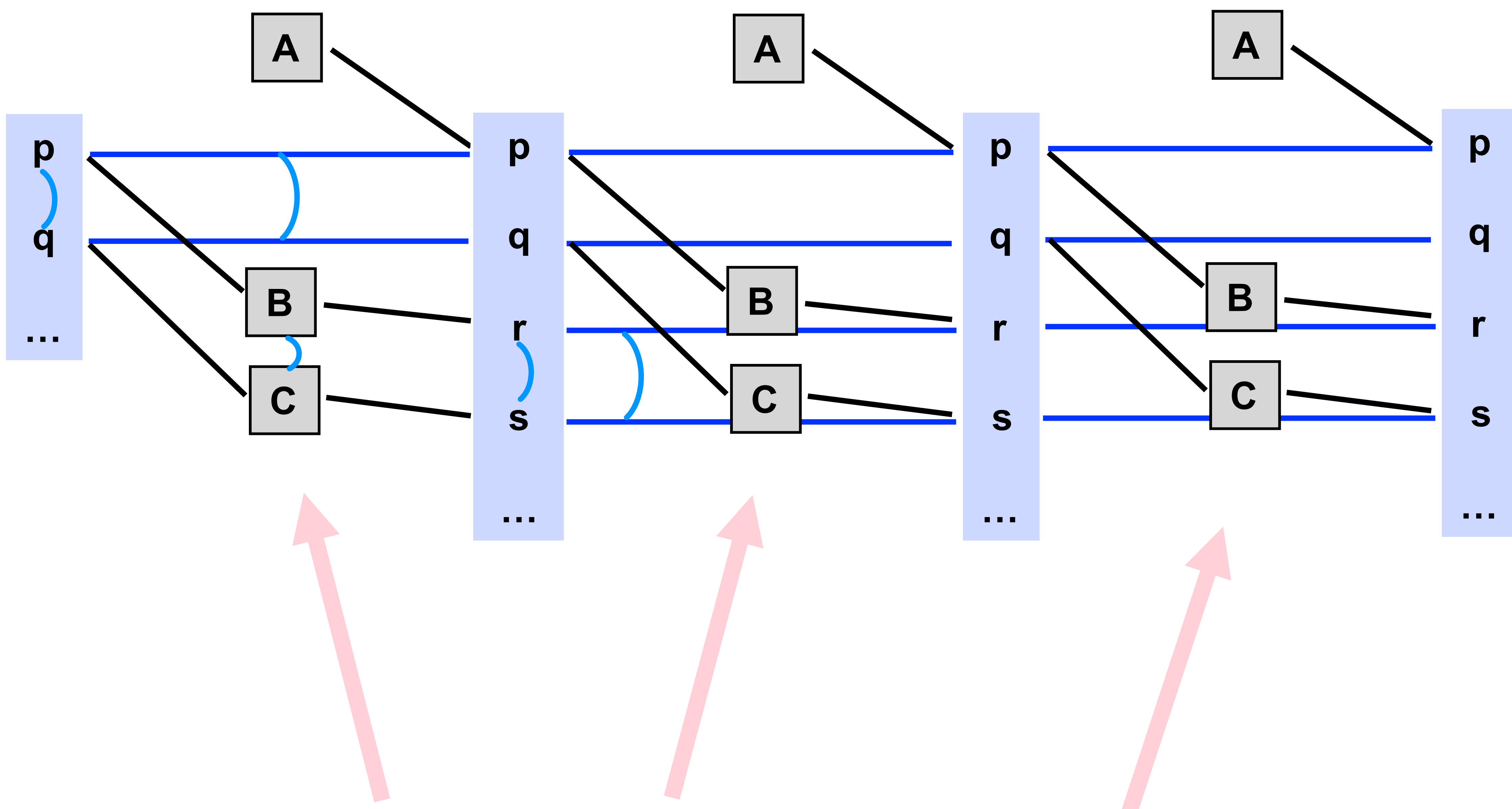
Actions monotonically increase

Properties 3



- Proposition mutex relationships monotonically decrease
- Specifically, if p and q are in layer n and are not mutex then they will not be mutex in future layers.

Properties 4



Action mutex relationships monotonically decrease

GraphPlan algorithm

- Grow the planning graph (PG) to a level n such that all goals are reachable and not mutex
  - necessary but insufficient condition for the existence of an n level plan that achieves the goals
  - if PG levels off before non-mutex goals are achieved then fail
- Search the PG for a valid plan
- If none found, add a level to the PG and try again
- If the PG levels off and still no valid plan found, then return failure

- Termination is guaranteed by the properties of the planning graph
- This termination condition does not guarantee completeness. Why?
- A more complex termination condition exists that does… (wait two slides)

Definition: Fix point of a planning graph

- **Definition 8**:
  - A planning graph has reached a fix point
    iff two fact levels and all their mutually exclusiveness relations are identical.
- **Theorem 1**:
  - If a planning graph has reached a fix point
    and if a partial goal is not contained in the final fact level
    or if two partial goals are mutually exclusive,
    then the planning problem has no solution.
- Must return failure.
- This condition is sufficient, but not necessary.

Iterative expansion and search

- If no valid plan can be extracted from the graph:
  - If fix point is not yet reached, expand graph with new level and search again.
- Is it possible that goals are not mutually exclusive, but plan extraction fails?
  - Yes! This occurs if an impossible state would have to be crossed on an intermediate level.
- **Theorem 2**:
  - Let i be the level on which the graph reaches its fix point. Let

    $$\left|G_i^t\right|$$

    be the number of unsolved subgoals on level i of a graph of depth t.

    A planning problem is unsolvable iff

    $$|G_i^t| = |G_i^{t+1}|$$

- This condition is sufficient and necessary!

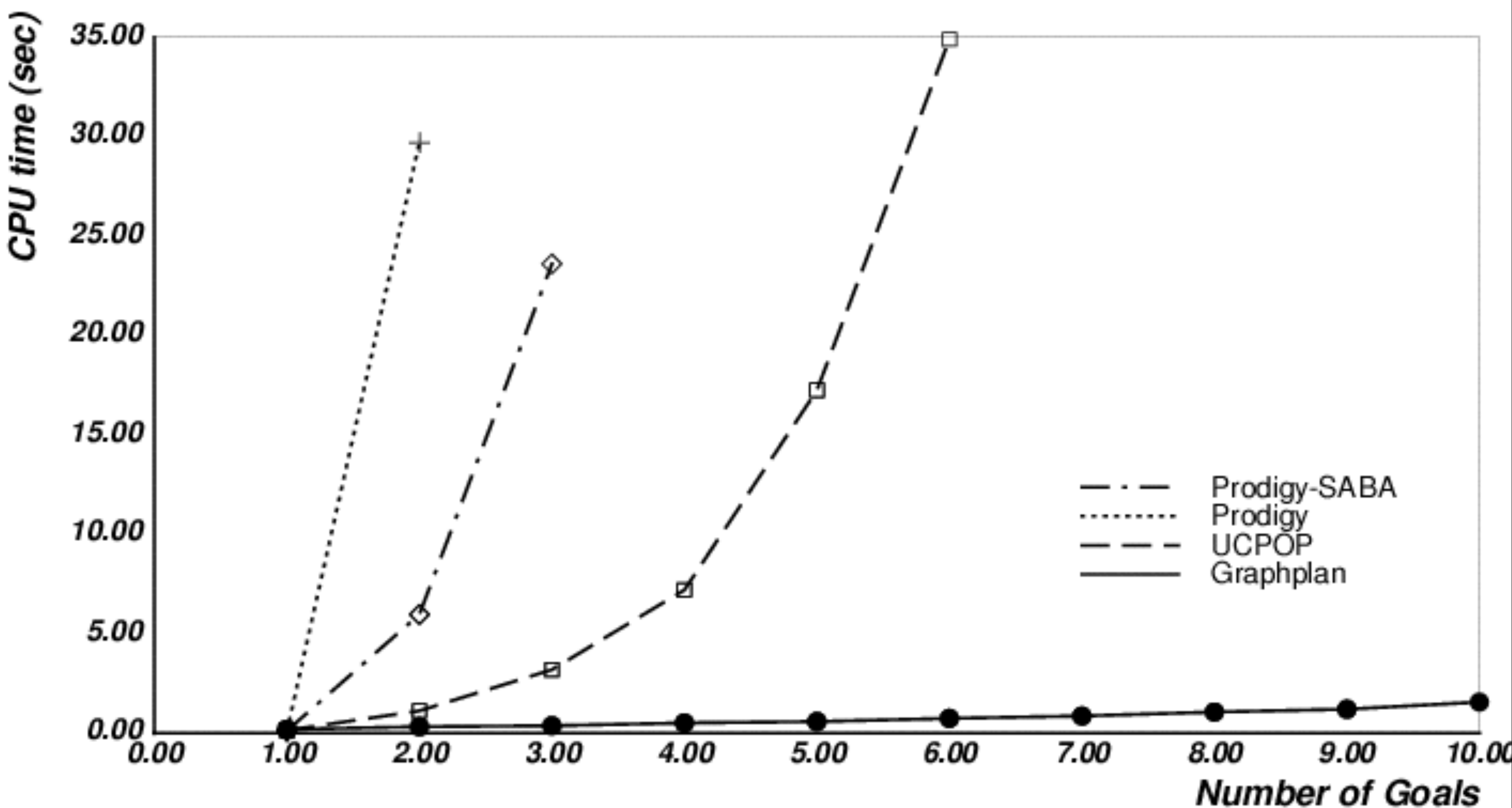
History and comparison with PSP

- 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
- Unlike PSP, GraphPlan creates ground instances of everything, many of them may be irrelevant
- But 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

Problem	Graph	Planning	Total
rocket-8	0.16	0	0.16
rocket-10	0.20	0	0.20

- Many subsequent planning systems have used ideas from it
  - IPP, STAN, GraphHTN, SGP, BlackBox, Medic, TGP, LPG
- Several of them are much faster than the original GraphPlan

Experimental comparison of GraphPlan



Why is GraphPlan so fast?

- Exclusiveness relations appear in all examples sufficiently often
- They can be propagated and heavily constrain the search space
- Many planning problems can be solved
  with partial-order or parallel plans,
  i.e. the planning graphs are not very deep
- Descriptions of planning problems to be solved
  rarely contain irrelevant information
  - Objects: letters not to be delivered
  - Facts: rooms never to be traversed during delivery
  - Actions: sorting or repackaging of postal items do not play a role