Automated Planning

Dana S. Nau

CMSC 421, Spring 2010

Some Dictionary Definitions of "Plan"

plan *n*.

1. A scheme, program, or method worked out beforehand for the accomplishment of an objective: *a plan of attack*.
2. A proposed or tentative project or course of action: *had no plans for the evening*.

- These two are closest to the meaning used in AI

3. A systematic arrangement of elements or important parts; a configuration or outline: *a seating plan; the plan of a story*.
4. A drawing or diagram made to scale showing the structure or arrangement of something.
5. A program or policy stipulating a service or benefit: *a pension plan*.

[a representation] of future behavior … usually a set of actions, with temporal and other constraints on them, for execution by some agent or agents. - Austin Tate

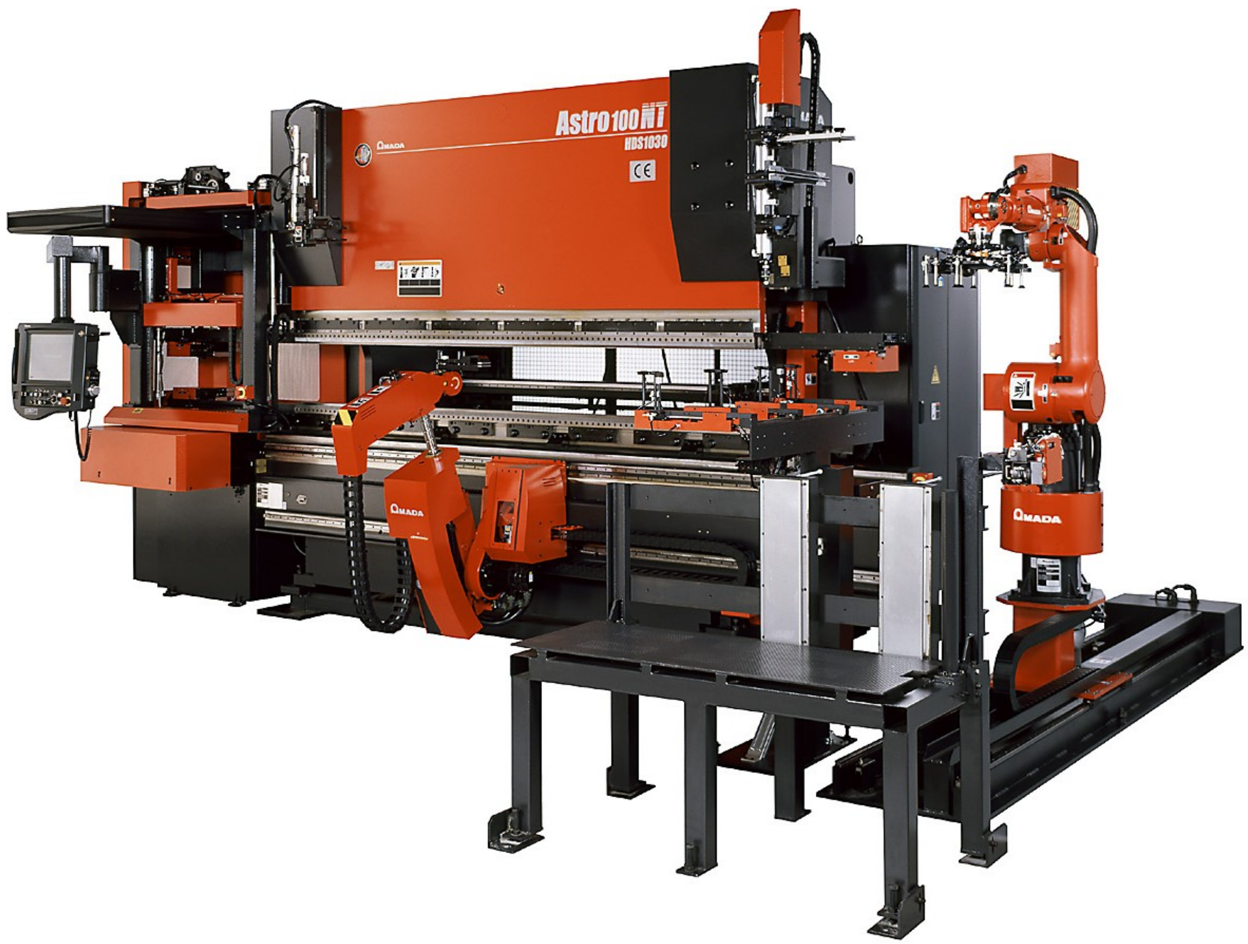
[*MIT Encyclopedia of the Cognitive Sciences*, 1999]

```
                               03  Establish datum point at bullseye (0.25, 1.00)
                               01  Install 0.15-diameter side-milling tool
                               02  Rough side-mill pocket at (-0.25, 1.25)
                                   length 0.40, width 0.30, depth 0.50
                               03  Finish side-mill pocket at (-0.25, 1.25)
                                   length 0.40, width 0.30, depth 0.50
                               04  Rough side-mill pocket at (-0.25, 3.00)
                                   length 0.40, width 0.30, depth 0.50
                               05  Finish side-mill pocket at (-0.25, 3.00)
                                   length 0.40, width 0.30, depth 0.50
                               01  Install 0.08-diameter end-milling tool
                               [...]
                               01  Total time on VMC1

                               01  Pre-clean board (scrub and wash)
                               02  Dry board in oven at 85 deg. F
005 B   EC1 30.00      0.48    01  Setup
                               02  Spread photoresist from 18000 RPM spinner
005 C   EC1 30.00      2.00    01  Setup
                               02  Photolithography of photoresist
                                   using phototool in "real.iges"
005 D   EC1 30.00     20.00    01  Setup
                               02  Etching of copper
005 T   EC1 90.00     54.77    01  Total time on EC1

006 A   MC1 30.00      4.57    01  Setup
                               02  Prepare board for soldering
006 B
006 C   MC1 30.00      7.50    01  Setup
```

A portion of a manufacturing process plan

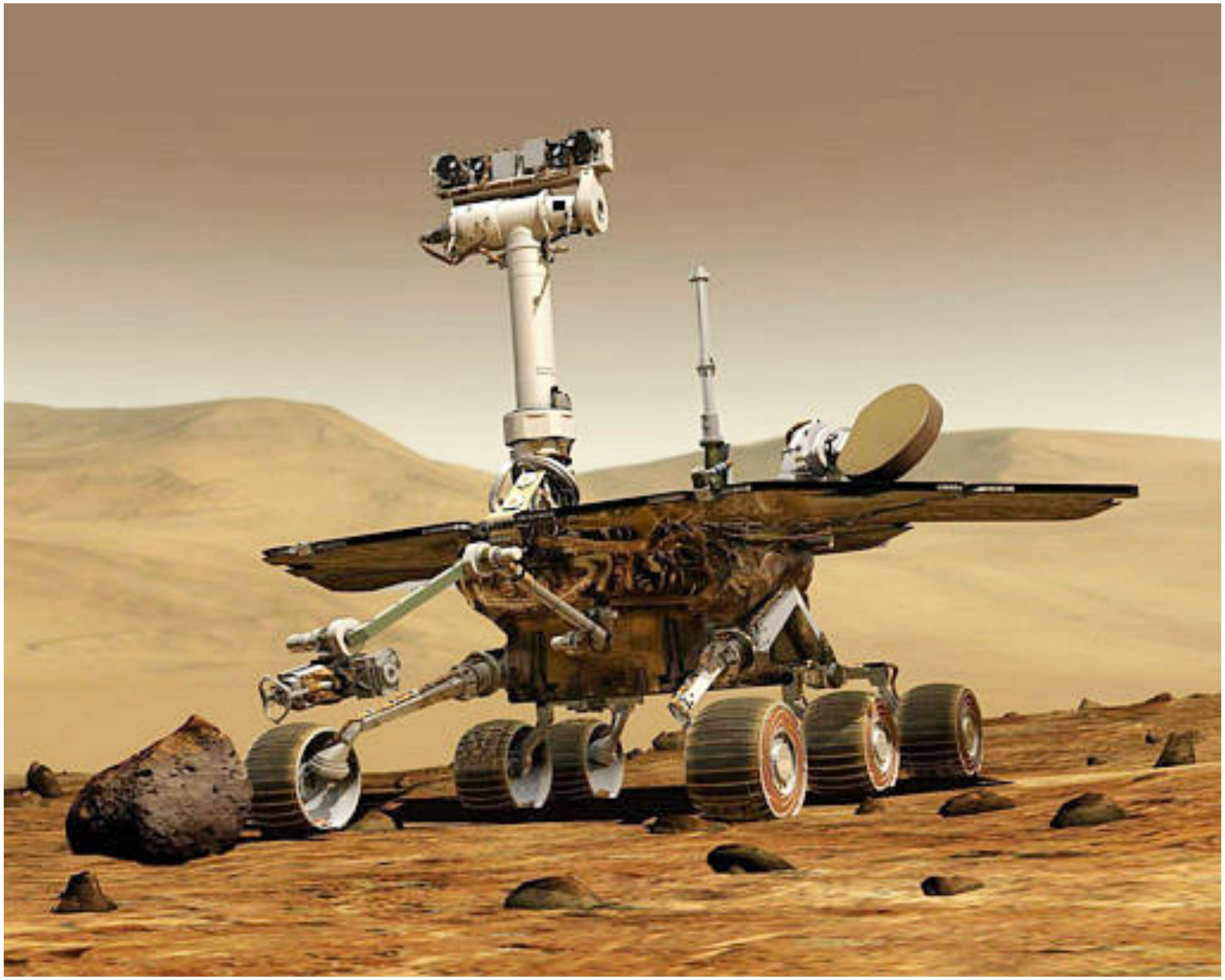
Manufacturing

- Sheet-metal bending machines - Amada Corporation
  - Software to plan the sequence of bends
    [Gupta and Bourne, *J. Manufacturing Sci. and Engr.*, 1999]



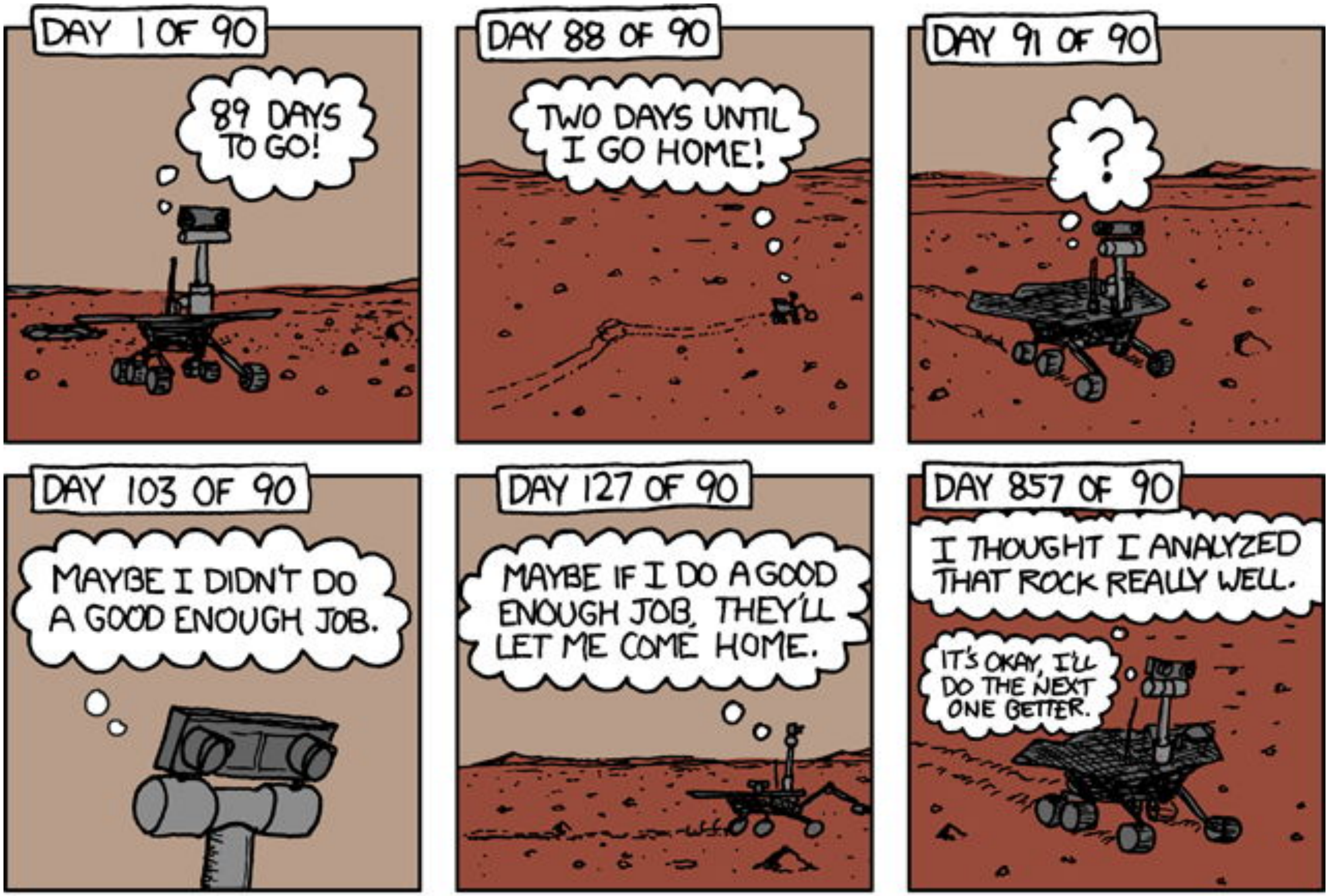
Space Exploration

- Autonomous planning, scheduling, control
  - NASA: JPL and Ames
- Remote Agent Experiment (RAX)
  - Deep Space 1
- Mars Exploration Rover (MER)



http://xkcd.com/695

- On January 26th, 2274 Mars days into the mission, NASA declared Spirit a 'stationary research station', expected to stay operational for several more months until the dust buildup on its solar panels forces a final shutdown.

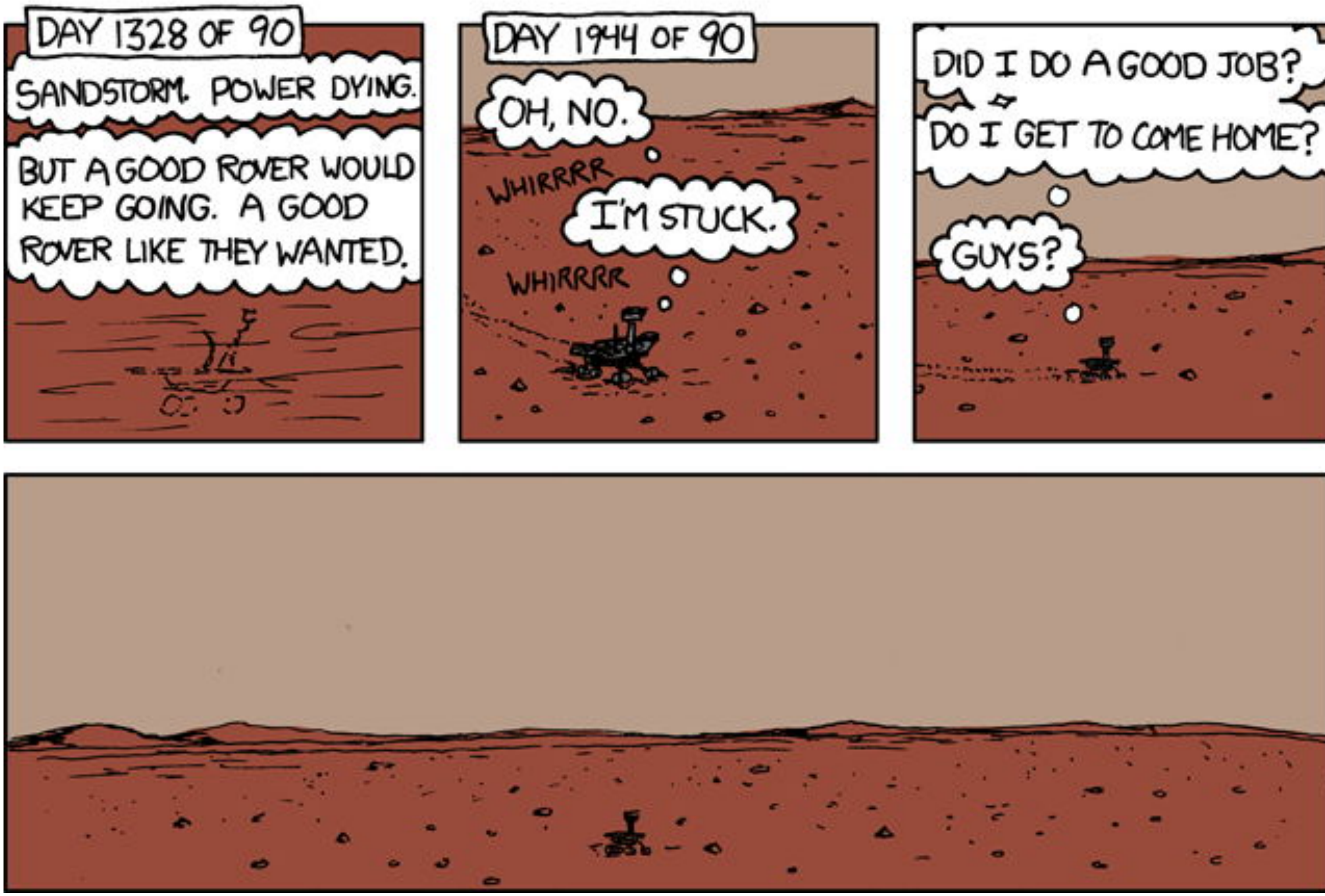


Continued on the next slide …

http://xkcd.com/695

- On January 26th, 2274 Mars days into the mission, NASA declared Spirit a 'stationary research station', expected to stay operational for several more months until the dust buildup on its solar panels forces a final shutdown.

Continued from the previous slide:



Outline

- Conceptual model for planning
- Restrictive assumptions to simplify the problem
- Classical planning

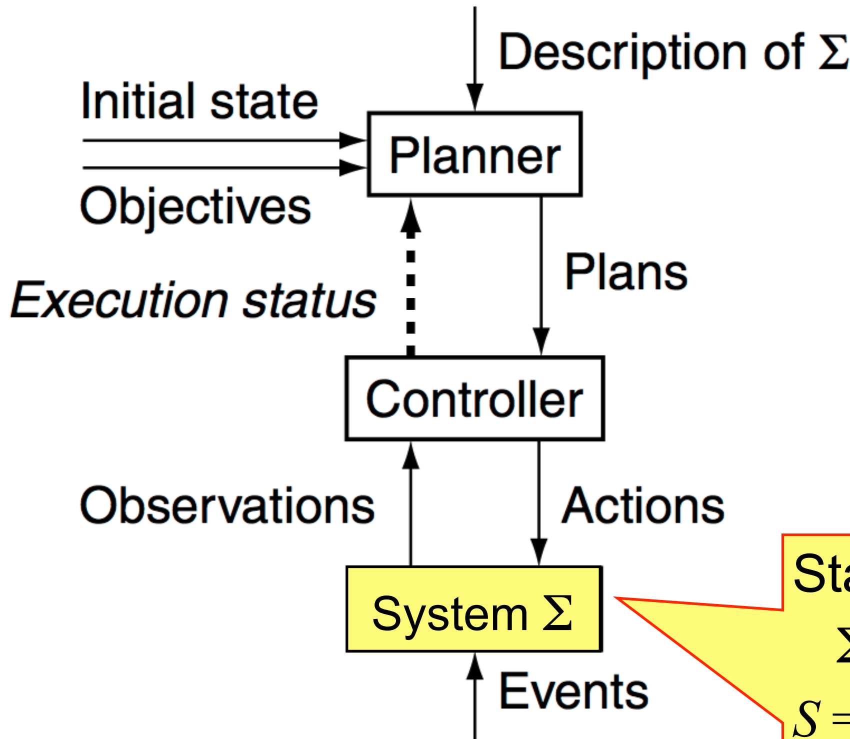
Source Material

- My lectures on AI planning are based partly on Russell & Norvig, and partly on following book:

- M. Ghallab, D. Nau, and P. Traverso
  Automated Planning: Theory and Practice
  Morgan Kaufmann Publishers
  May 2004
  - Web site: http://www.laas.fr/planning

- For CMSC 421, you ***don't*** need this book
  - The lecture slides are self-contained

Conceptual Model
1. Environment



State transition system
$\Sigma = (S, A, E, \gamma)$

S = {states}

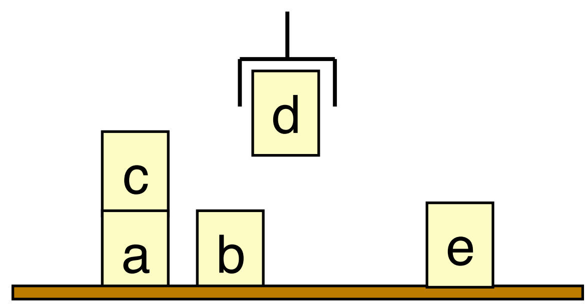
A = {actions}

E = {exogenous events}

γ = state-transition function

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:

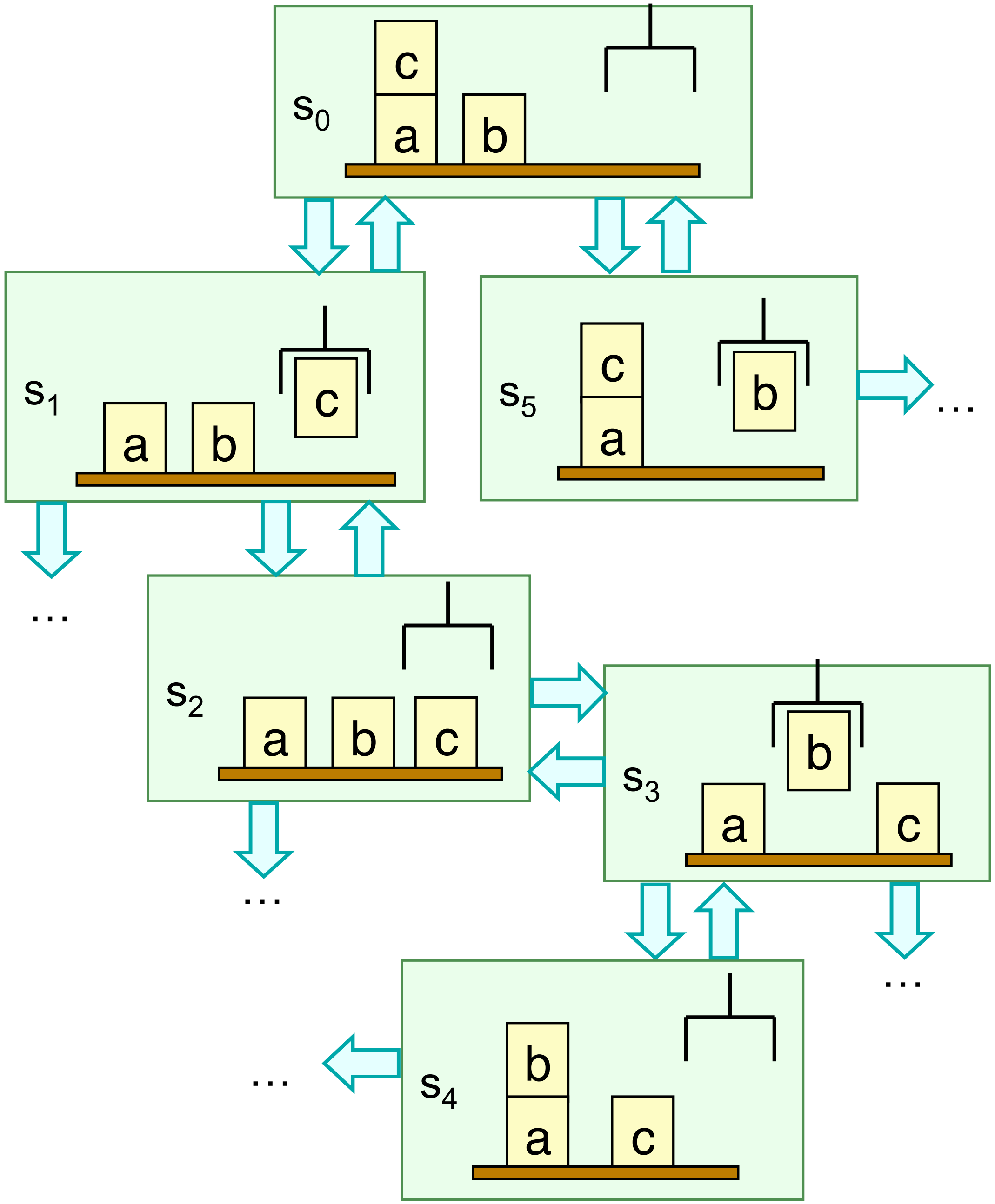


- For n blocks, the number of states is more than n!

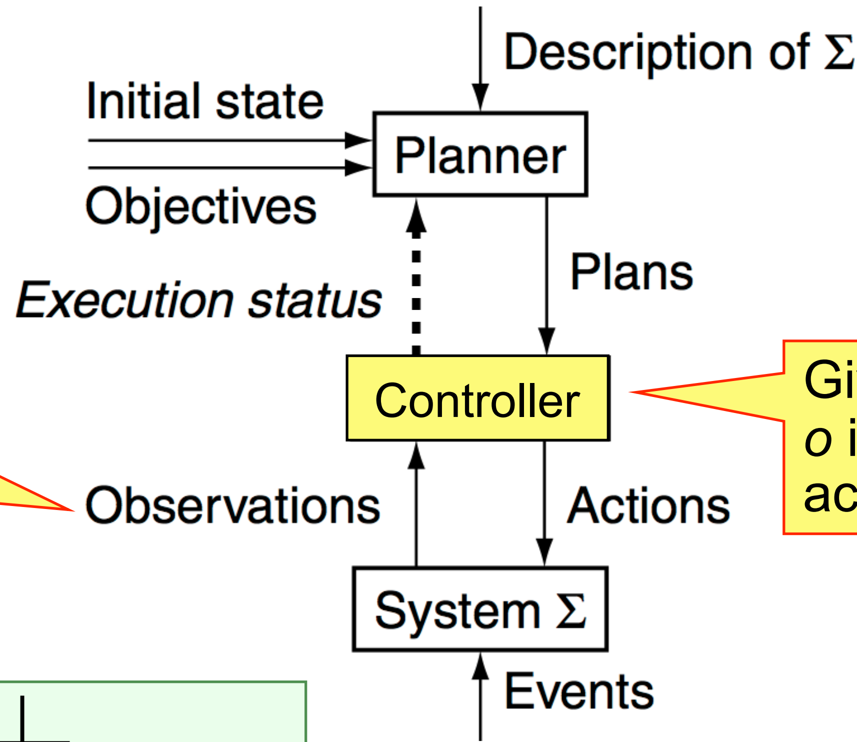
State Transition System

$\Sigma = (S,A,E,\gamma)$

- S = {states}
- A = {actions}
- E = {exogenous events}
- State-transition function
  $\gamma: S \times (A \cup E) \rightarrow 2^S$
  - $S = \{s_0, s_1, s_2, \ldots, s_{22}\}$
  - A = {take c off of a,
    put c on the table,
    …}
  - $E = \{\}$
  - γ: see the arrows

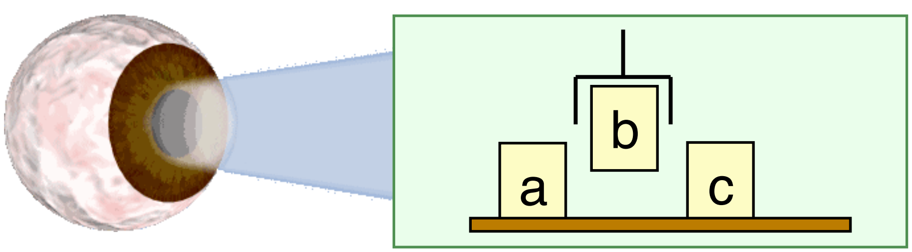


Conceptual Model
2. Controller

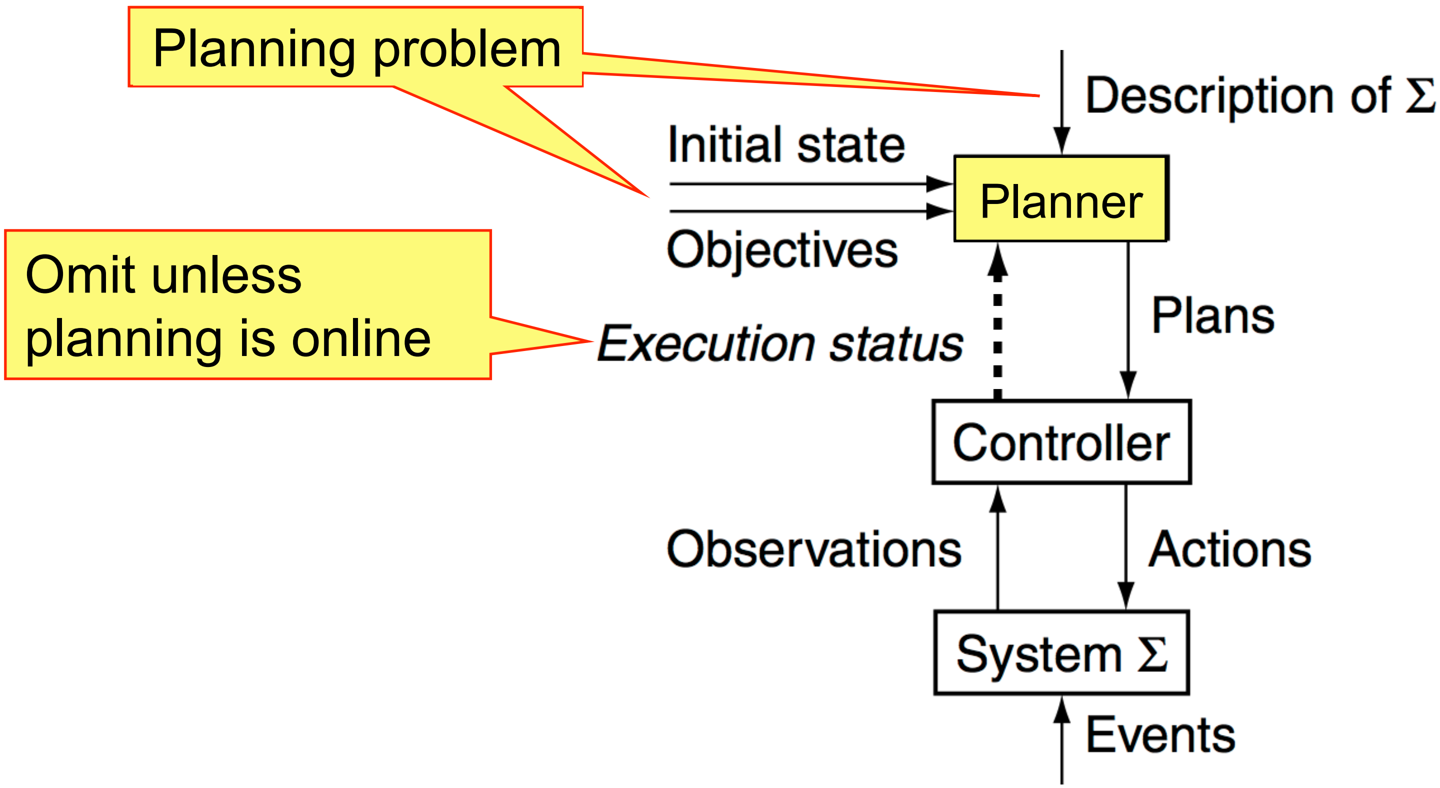


Given observation *o* in *O*, produces action *a* in *A*

Observation function
$h: S \to O$



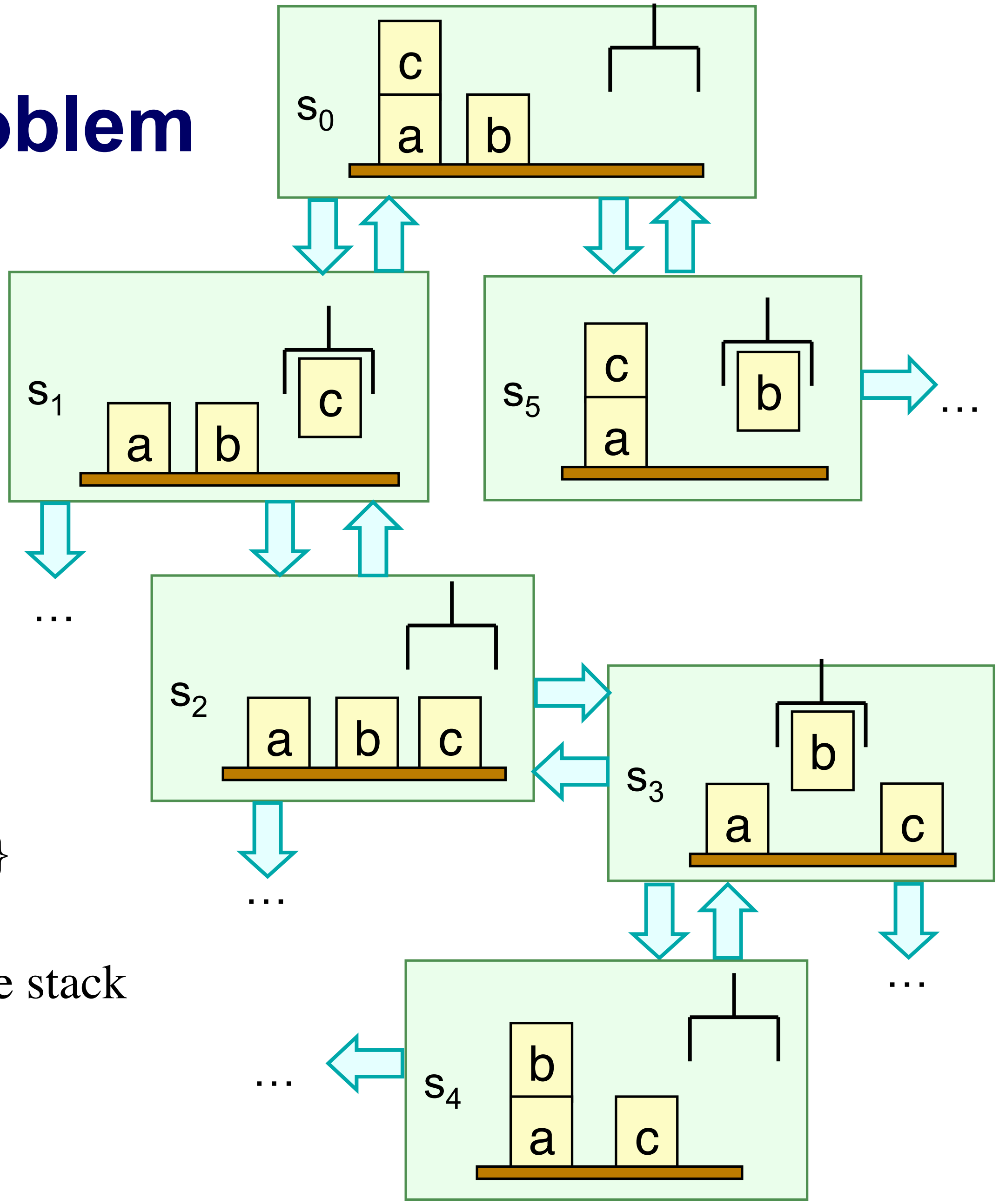
Conceptual Model
3. Planner's Input



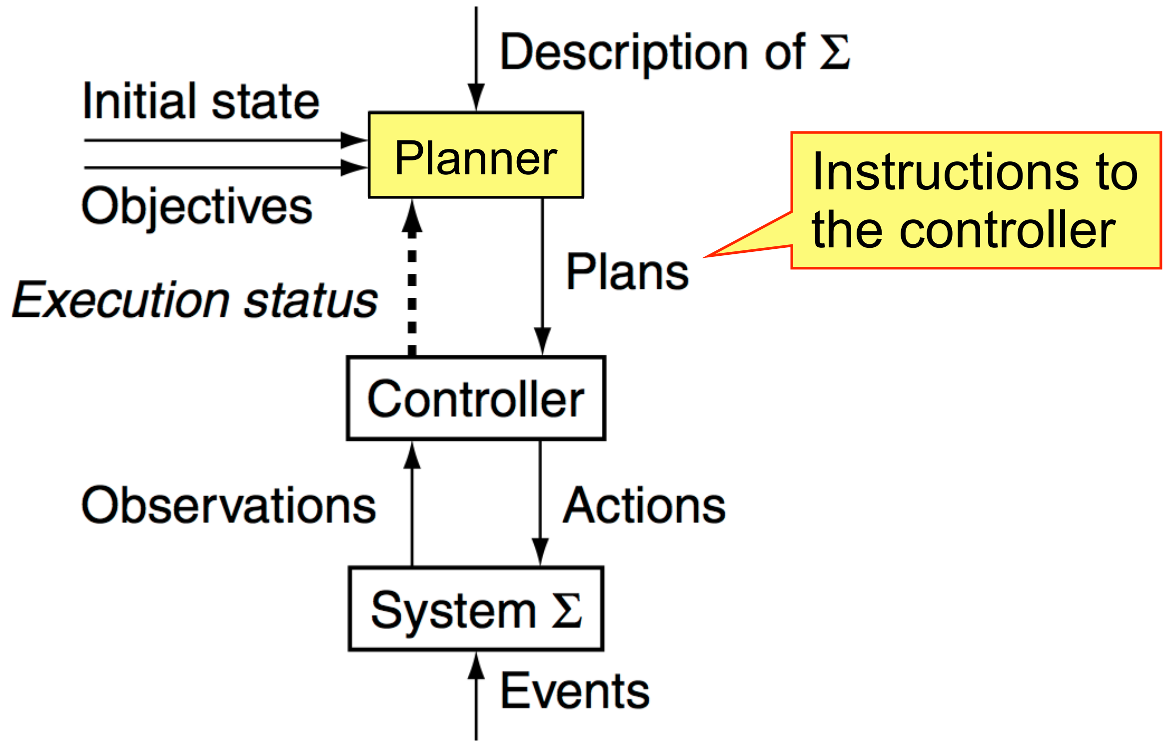
Planning Problem

A planning problem includes:

- A description of Σ
- An initial state, e.g., s_0
  - or a set of possible initial states (maybe with a probability distribution)
- An objective, e.g.,
  - a goal state, e.g., s_4
  - a set of goal states, e.g., {all states in which b is on a}
  - a task to perform, e.g., put all the blocks into a single stack
  - a "trajectory" of states
  - an objective function
  - …



Conceptual Model
4. Planner's Output



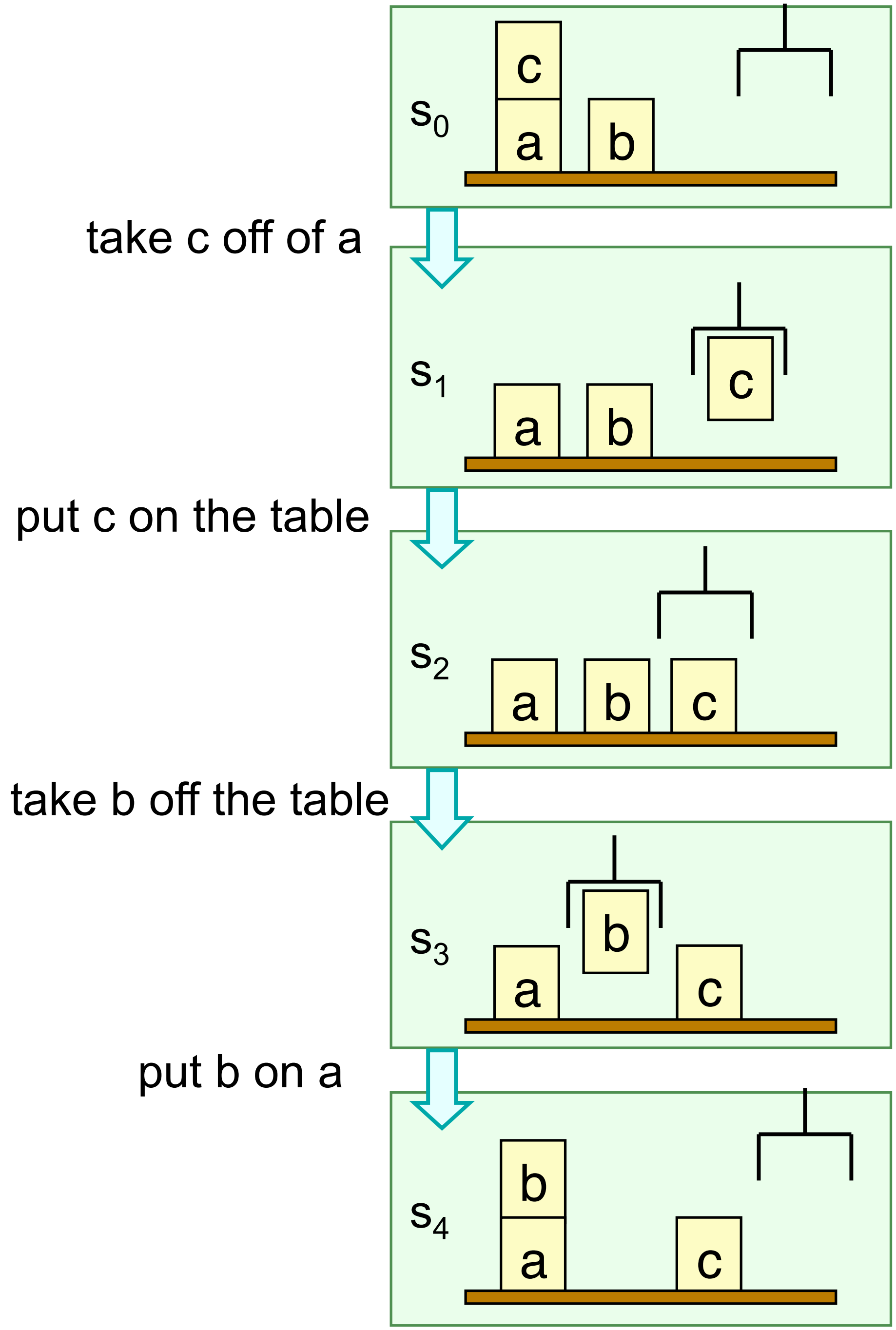
Plans

- **Classical plan**:
  a sequence of actions

  (take c off of a,
  put c on the table,
  take b off the table,
  put b on a)

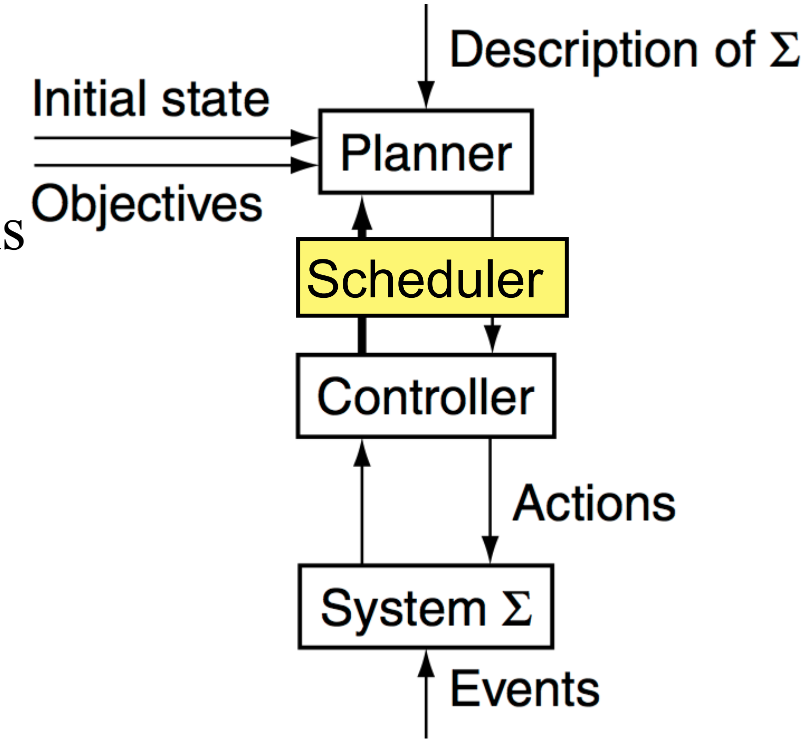
- **Policy**:
  a partial function from *S* into *A*

  {(s_0, take c off of a),
  (s_1, put c on the table),
  (s_2, take b off the table),
  (s_3, put b on a)}



Planning Versus Scheduling

- Scheduling
  - Decide when and how to perform a given set of actions
    - » Time constraints
    - » Resource constraints
    - » Objective functions
  - Typically NP-complete



- Planning
  - Decide what actions to use to achieve some set of objectives
  - Can be much worse than NP-complete
    - » worst case is undecidable

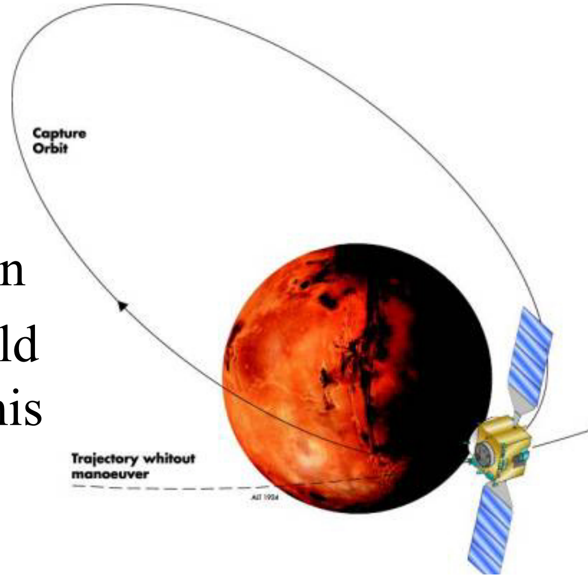
Three Main Types of Planners

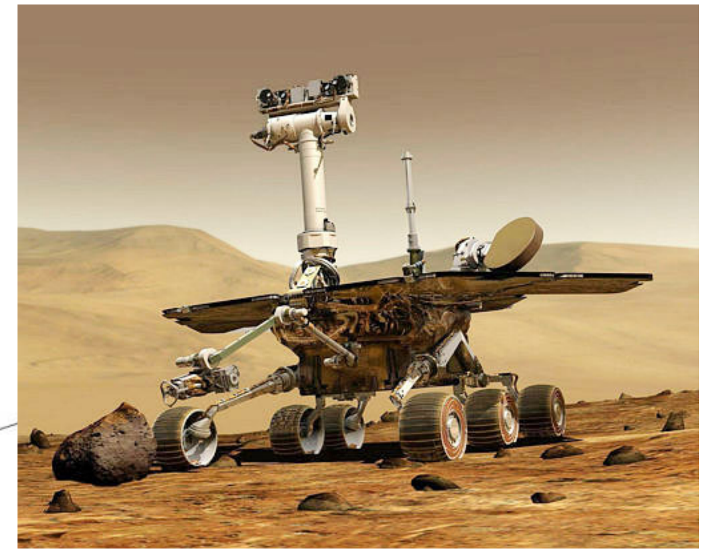
1. Domain-specific
2. Domain-independent
3. Configurable

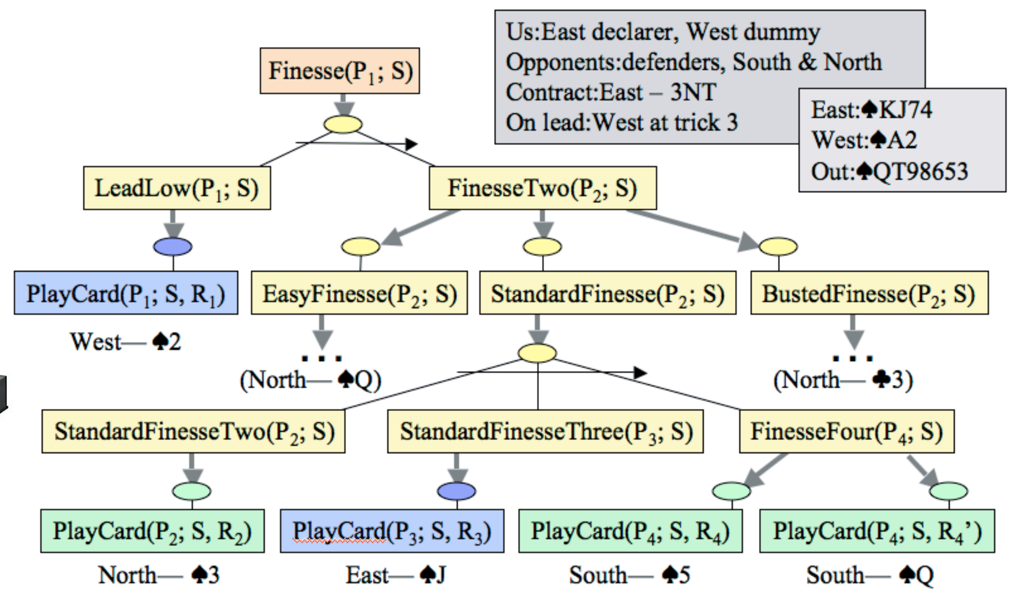
- I'll talk briefly about each

1. Domain-Specific Planners (Chapters 19-23)

- Made or tuned for a specific domain
- Won’t work well (if at all) in any other domain
- Most successful real-world planning systems work this way



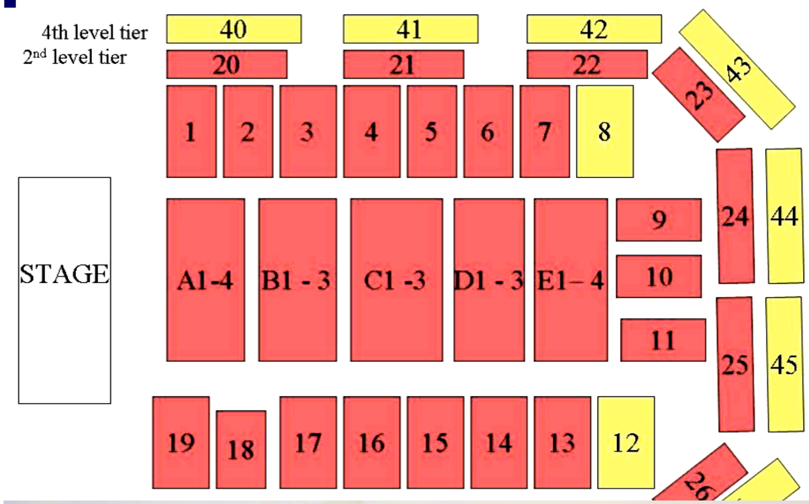




Types of Planners 2. Domain-Independent

- In principle, a domain-independent planner works in any planning domain
- Uses no domain-specific knowledge except the definitions of the basic actions

Types of Planners
2. Domain-Independent

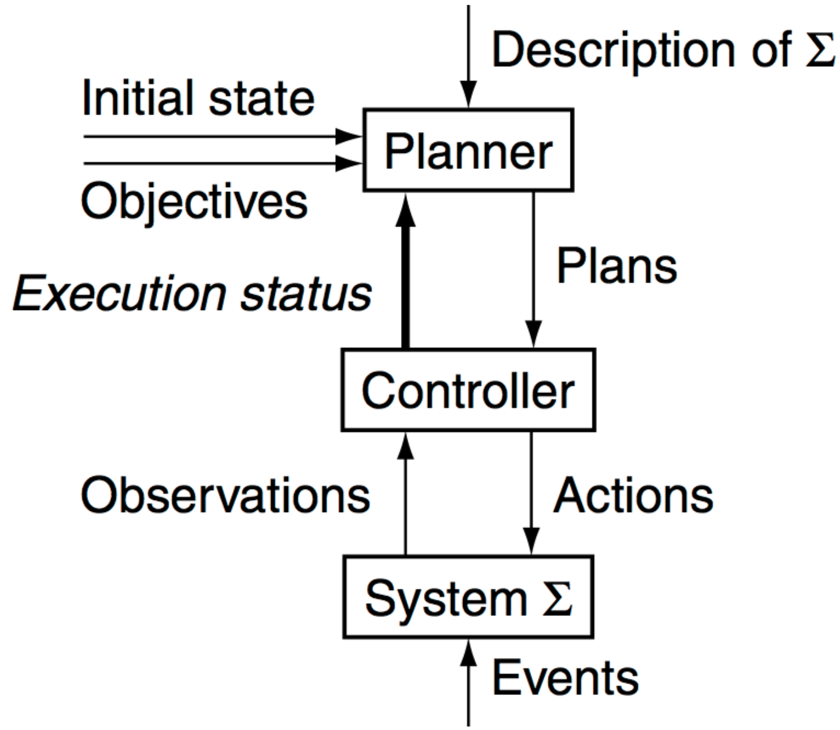


- In practice,
  - Not feasible to develop domain-independent planners that work in *every* possible domain

- Make simplifying assumptions to restrict the set of domains
  - *Classical planning*
  - Historical focus of most automated-planning research

Restrictive Assumptions

- **A0: Finite system:**
  - finitely many states, actions, events
- **A1: Fully observable:**
  - the controller always Σ's current state
- **A2: Deterministic:**
  - each action has only one outcome
- **A3: Static** (no exogenous events):
  - no changes but the controller's actions
- **A4: Attainment goals:**
  - a set of goal states S_g
- **A5: Sequential plans:**
  - a plan is a linearly ordered sequence of actions (a_1, a_2, ... a_n)
- **A6: Implicit time:**
  - no time durations; linear sequence of instantaneous states
- **A7: Off-line planning:**
  - planner doesn't know the execution status



Classical Planning (Chapters 2-9)

- Classical planning requires all eight restrictive assumptions
  - Offline generation of action sequences for a deterministic, static, finite system, with complete knowledge, attainment goals, and implicit time
- Reduces to a search problem:
  - Given (Σ, s_0, S_g)
    - s_0 is the initial state, S_g is a set of goal states
  - Find a sequence of actions $(a_1, a_2, \ldots a_n)$ that produces a sequence of state transitions $(s_1, s_2, \ldots, s_n)$ such that s_n is in S_g.
- Constraint-satisfaction problems also were search problems
  - But there were special-purpose problem representations and algorithms that were much faster than ordinary search algorithms
- Can do something similar for planning problems
  - Several ways to do this
  - I'll discuss a few of the better-known ones

Problem Representation

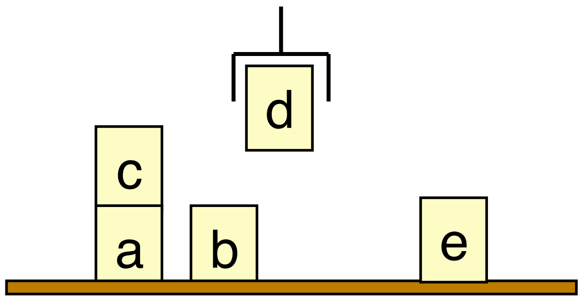
- Several ways to represent classical planning domains
  - The *classical representation* (or *STRIPS representation*) is the best known
- That's what I'll describe

Symbols

- Start with a *function-free* first-order language
  - Finitely many predicate names and constant symbols, infinitely many variable symbols, but *no* function symbols
  - Add a finite set of *operator names*
- e.g., symbols for the blocks world:
  - Constant symbols: a, b, c, d, e, … (names of blocks)
  - Variable symbols: $u, v, w, x, y, z, x_1, x_2, \ldots$
  - 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

  - Operator names: pickup, putdown, stack, unstack

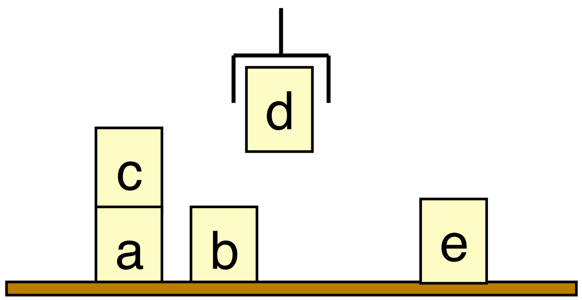


States

- State: a set *s* of ground atoms representing what's currently true
- Only finitely many ground atoms, so only finitely many possible states

- Example:

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



Operators

- *Operator*: a triple (head, preconditions, effects)
  - head: an operator name and a parameter list
    - » E.g., opname(x_1, …, 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($\boldsymbol{x_1}$, …, $\boldsymbol{x_k}$)**
    - » Precond: $p_1, p_2, \ldots, p_m$
    - » Effects: $e_1, e_2, \ldots, e_n$

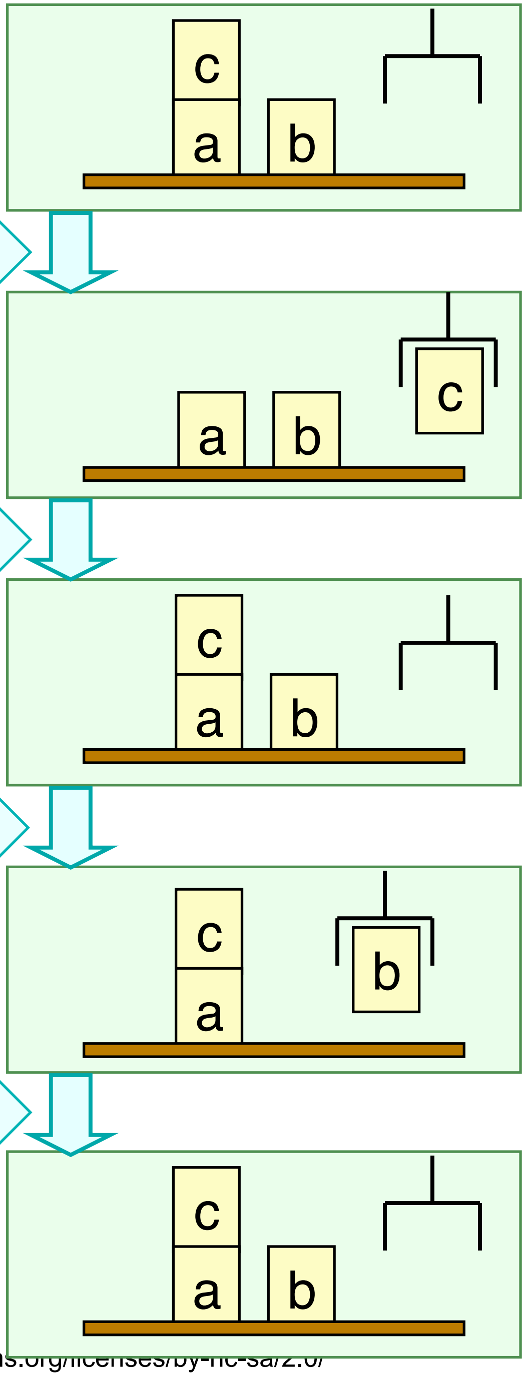
Blocks-World Operators

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

stack(x,y)
Precond: holding(x), clear(y)
Effects: ~holding(x), ~clear(y), on(x,y), clear(x), handempty

pickup(x)
Precond: ontable(x), clear(x), handempty
Effects: ~ontable(x), ~clear(x), ~handempty, holding(x)

putdown(x)
Precond: holding(x)
Effects: ~holding(x), ontable(x), clear(x), handempty

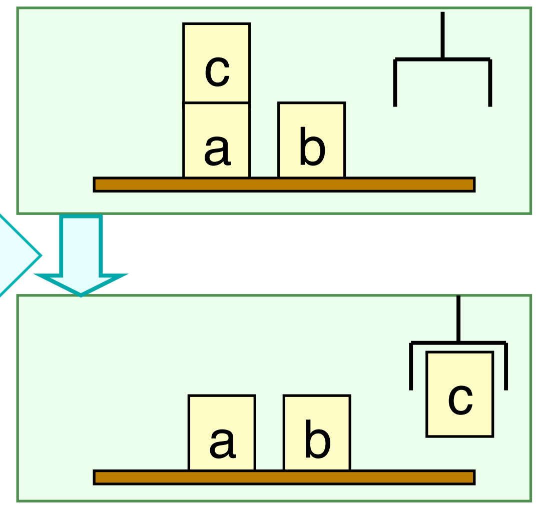


Actions and Plans

- 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
  - precond$^+(a)$ = {atoms that appear positively in precond(a)}
  - precond$^-(a)$ = {atoms that appear negatively in precond(a)}
  - effects$^+(a)$ = {atoms that appear positively in effects(a)}
  - effects$^-(a)$ = {atoms that appear negatively in effects(a)}
- Example:

  unstack(x,y)

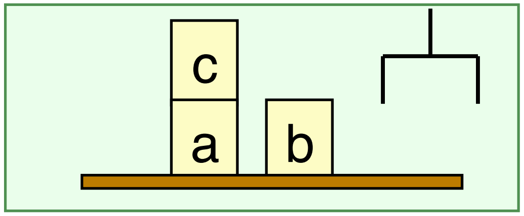
  Precond: on(x,y), clear(x), handempty

  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 precond$^+(a) \subseteq s$ and precond$^-(a) \cap s = \emptyset$
- An operator o is *applicable* to s if there's a ground instance a of o that is executable in s
- Example:
- s = {ontable(a), on(c,a), clear(c), ontable(b), clear(b),handempty}
- o = unstack(x,y)
- a = unstack(c,a)



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)

Result of performing an action

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

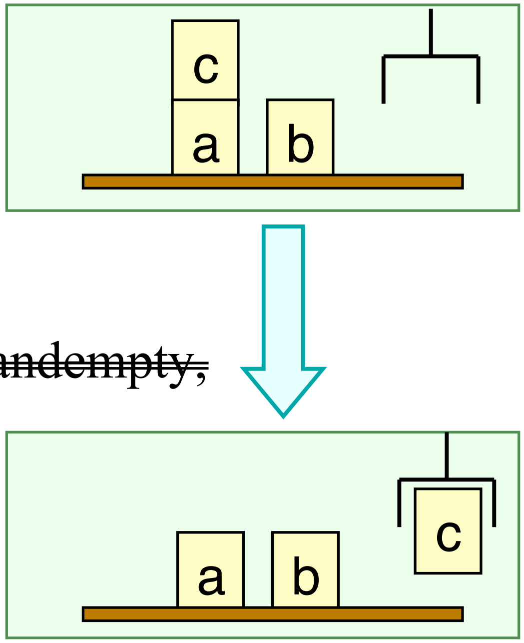
  $$\gamma(s,a) = (s - \text{effects}^{-}(a)) \cup \text{effects}^{+}(a)$$

  - Delete the negative effects, and add the positive ones

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

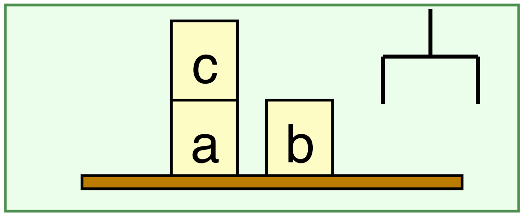
- a = unstack(c,a)
  Precond: on(c,a), clear(c), handempty
  Effects: ~on(c,a), ~clear(c), ~handempty,
  holding(c), clear(a)

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



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}

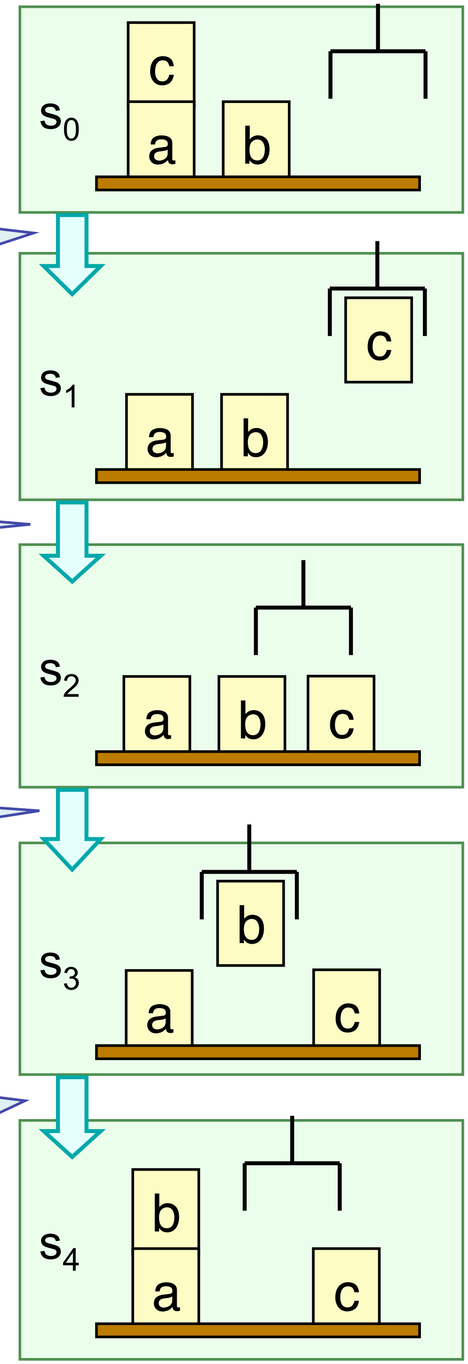
π = (unstack(c,a), putdown(c), pickup(b), stack(b,a))

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

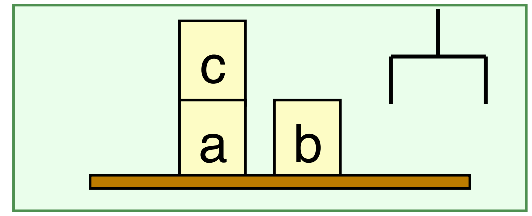
- *Planning problem*: a triple $P = (O, s_0, g)$
  - O is a set of operators
  - s_0 is the *initial state* - a set of atoms
  - g the *goal formula* - a set of literals
- Every state that satisfies g is a *goal state*

- 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

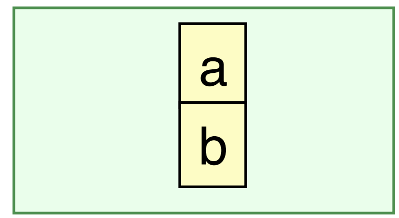
Example

- 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}

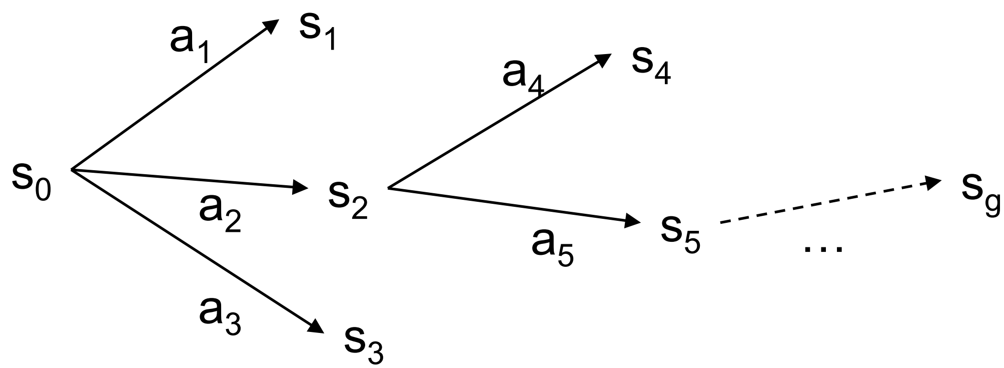


- g = {on(a,b)}



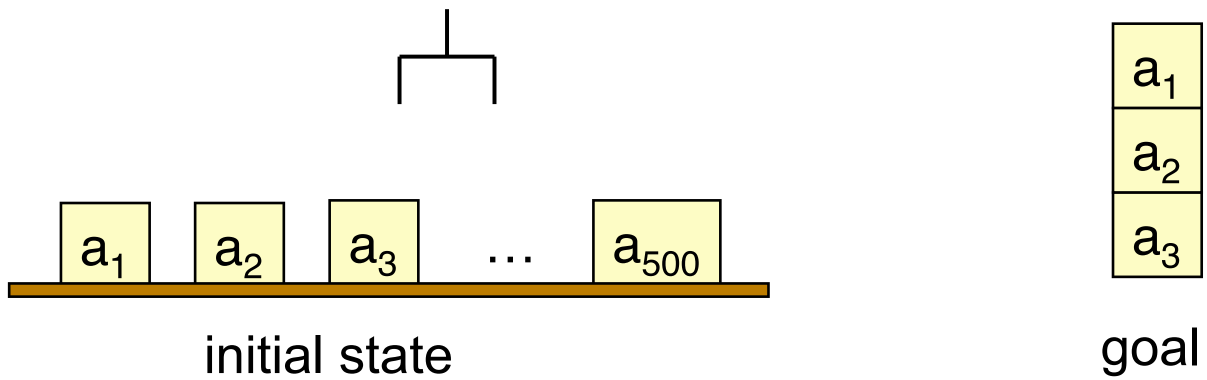
- One of the solutions is
  - π = (unstack(c,a), putdown(c), pickup(a), stack(a,b))

Forward-Search Algorithms



- 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

Branching Factor of Forward Search



- Forward search can have a very large branching factor
  - pickup(a_1), pickup(a_2) , …, pickup(a_{500})
- Thus forward-search can waste time trying lots of irrelevant actions
  - Need a good heuristic to guide the search
  - I'll discuss one later
- But first, a very different kind of planning algorithm

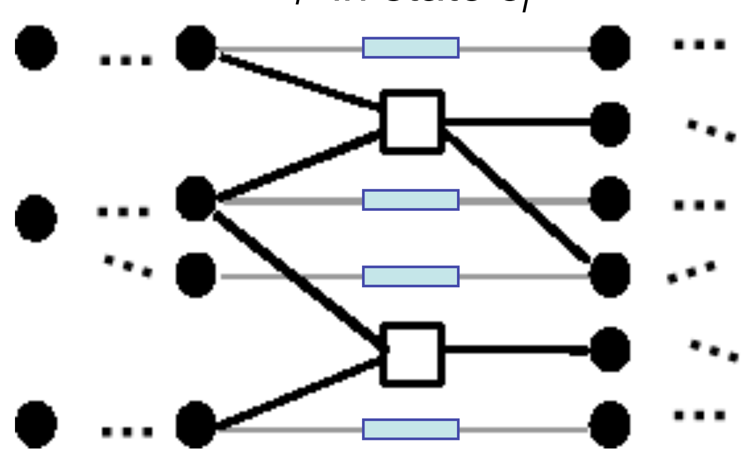
Graphplan

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

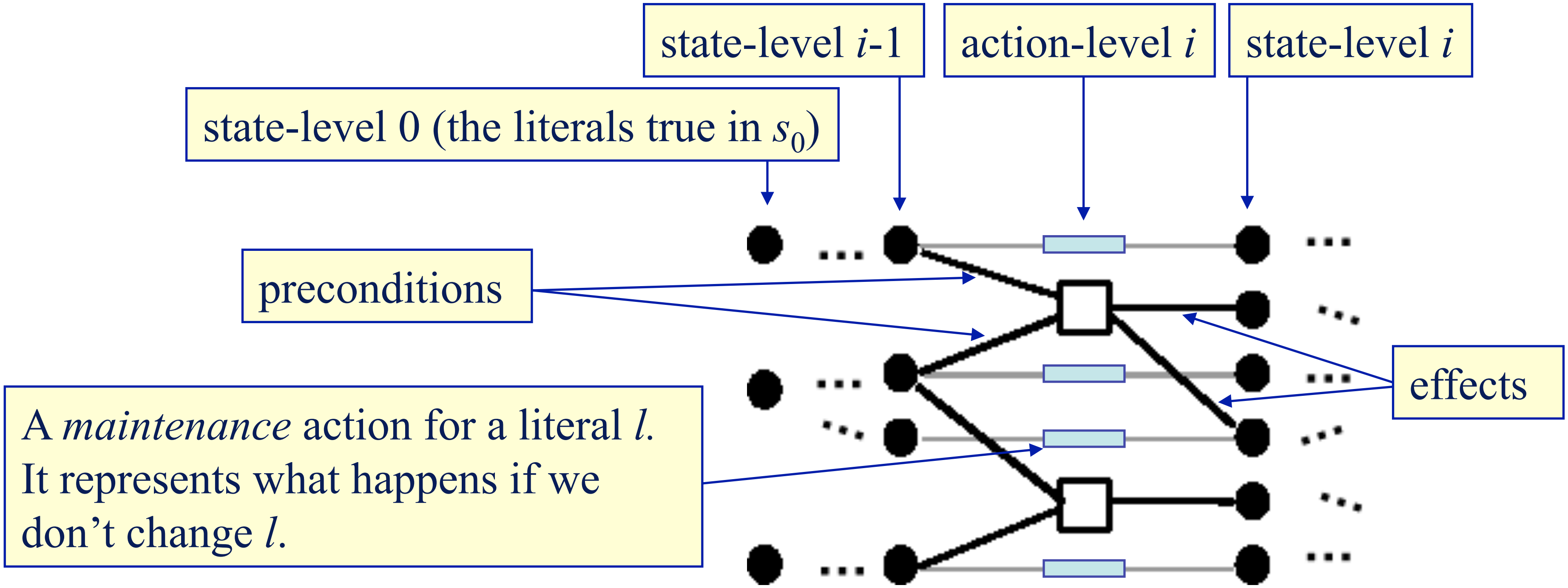
relaxed problem

  - 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
  - At action-level i: all actions whose preconditions appear in state-level i–1
  - At state-level i: all the effects of all the actions at action-level 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

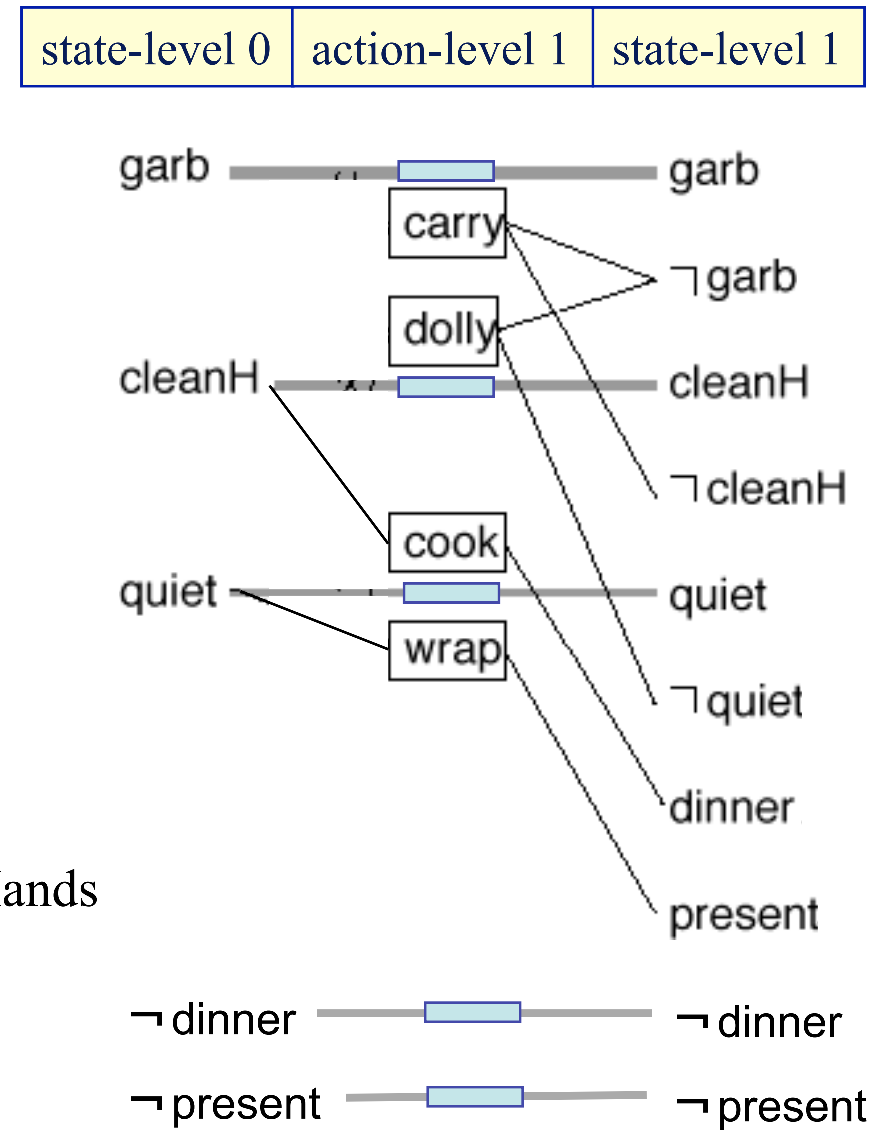
Also have the maintenance actions: one for each literal

Example (continued)

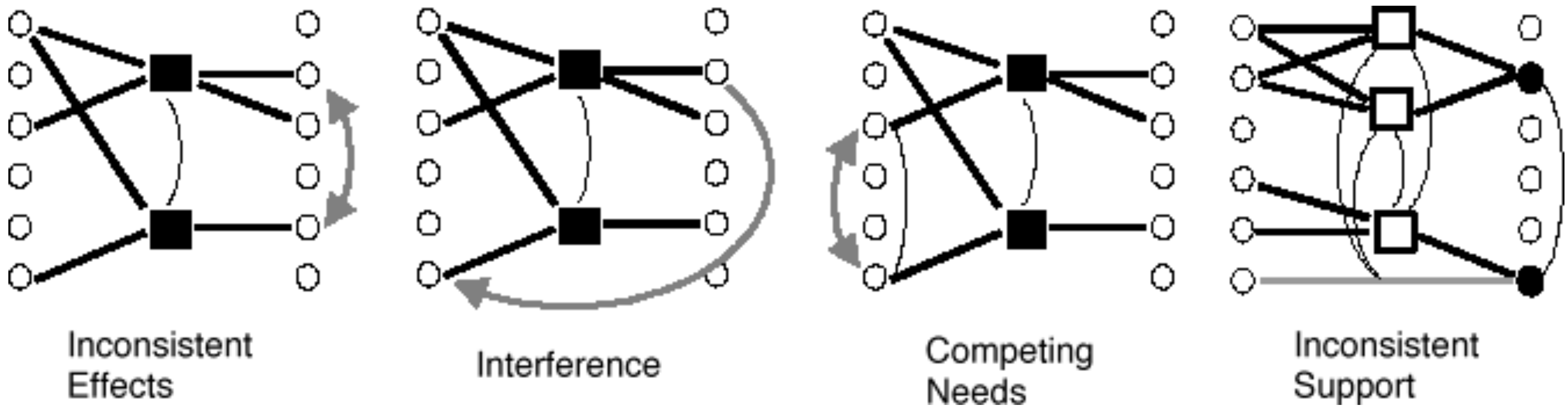
- state-level 0:
  {all atoms in s_0} U
  {negations of all atoms not in s_0}
- action-level 1:
  {all actions whose preconditions
  are satisfied and non-mutex in s_0}
- state-level 1:
  {all effects of all of the
  actions in action-level 1}

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**

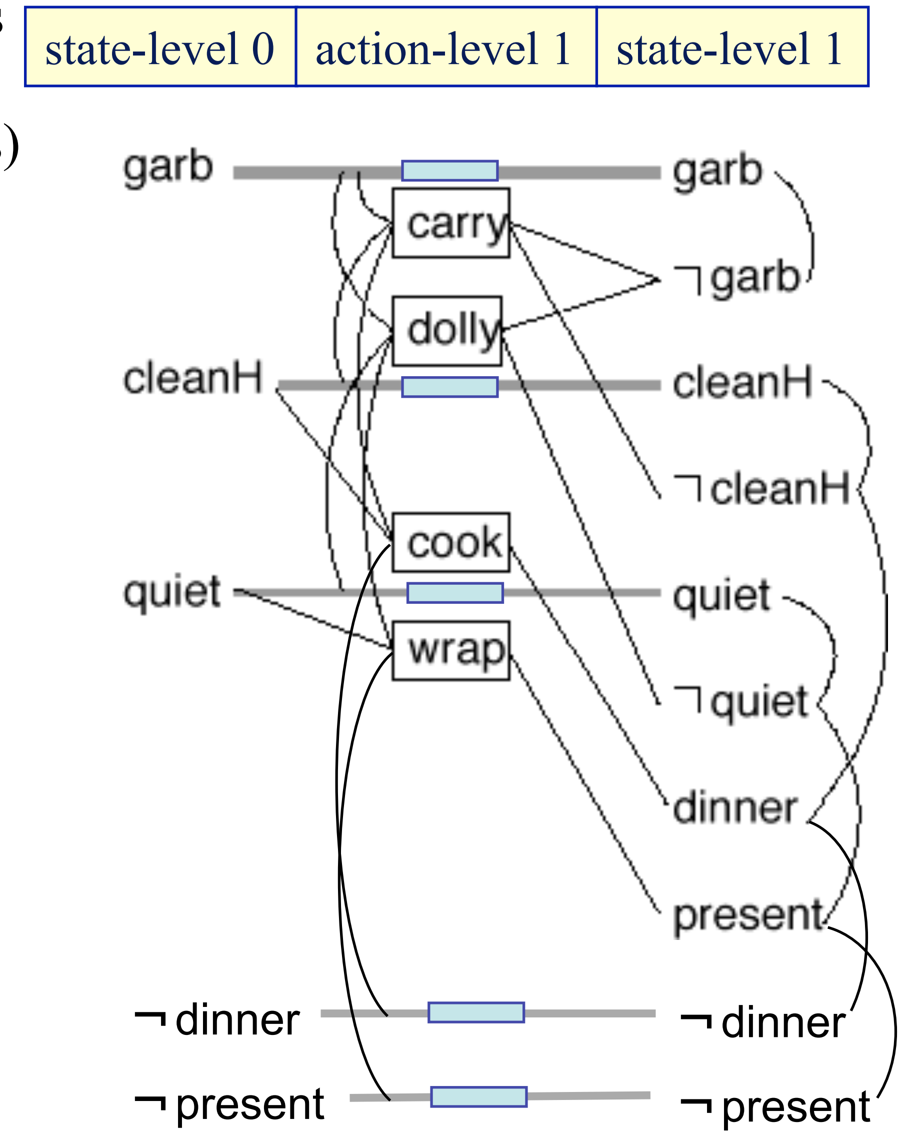
Recursive propagation of mutexes

Example (continued)

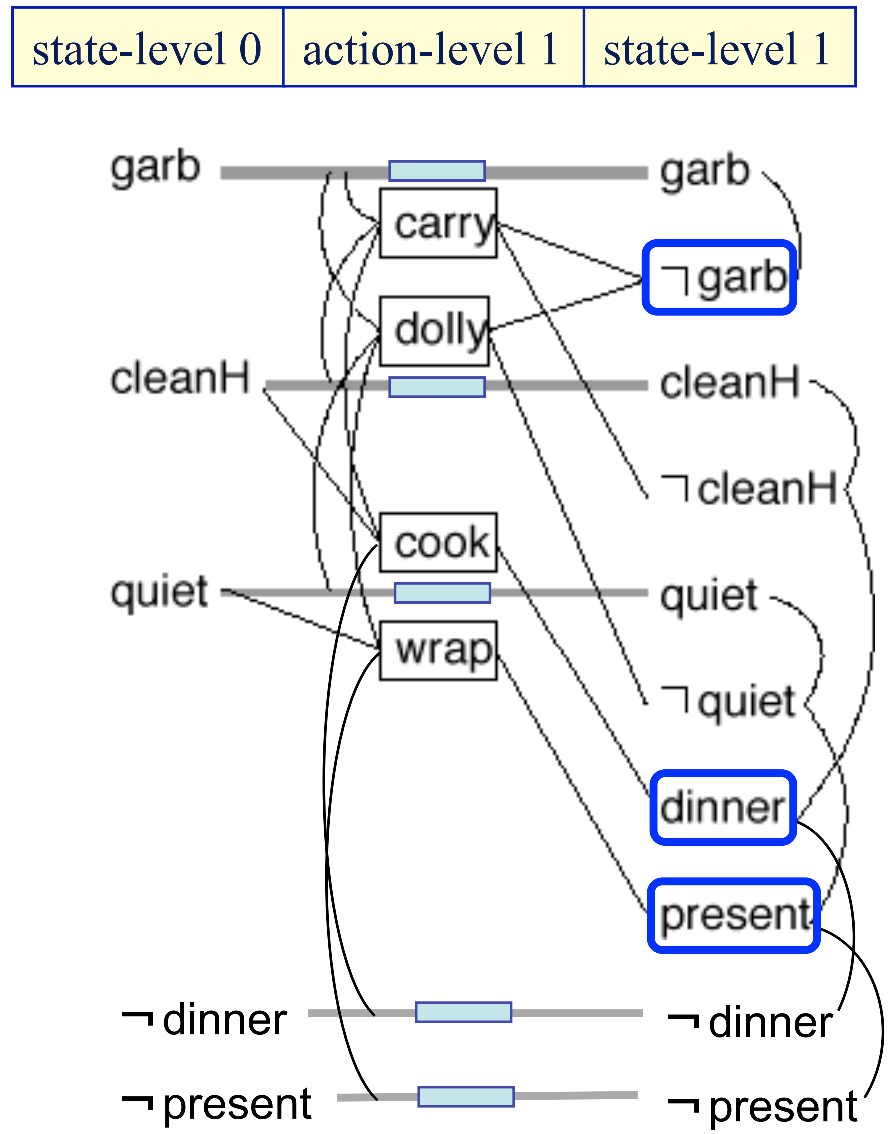
- Augment the graph to indicate mutexes
- *carry* is mutex with the maintenance action for *garbage* (inconsistent effects)
- *dolly* is mutex with *wrap*
  - interference
- ~*quiet* is mutex with *present*
  - inconsistent support
- each of *cook* and *wrap* is mutex with a maintenance operation

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

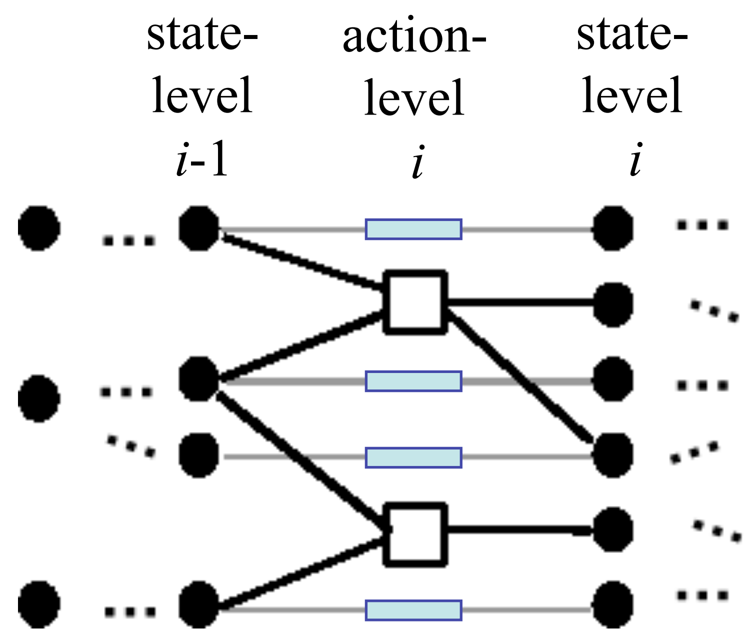
if any pair of chosen actions are mutex

then backtrack

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

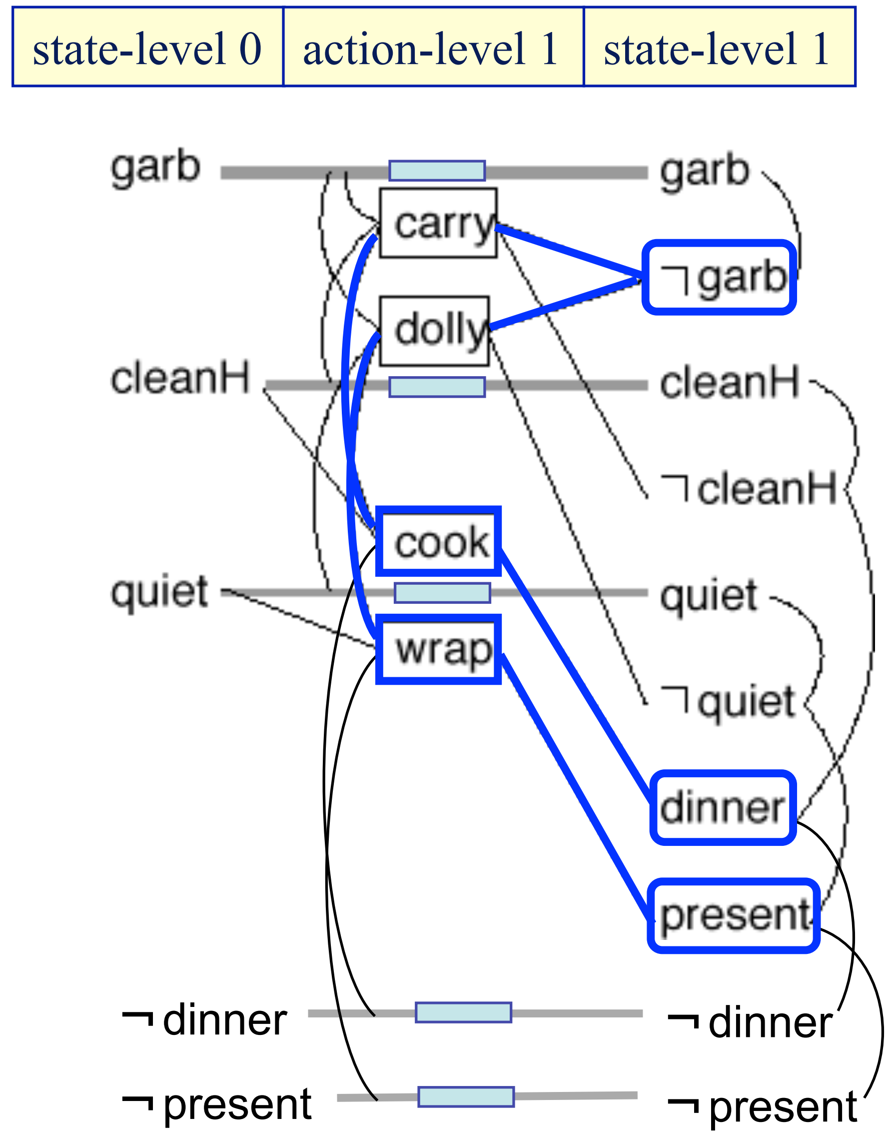
Solution-extraction(g', j–1)

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



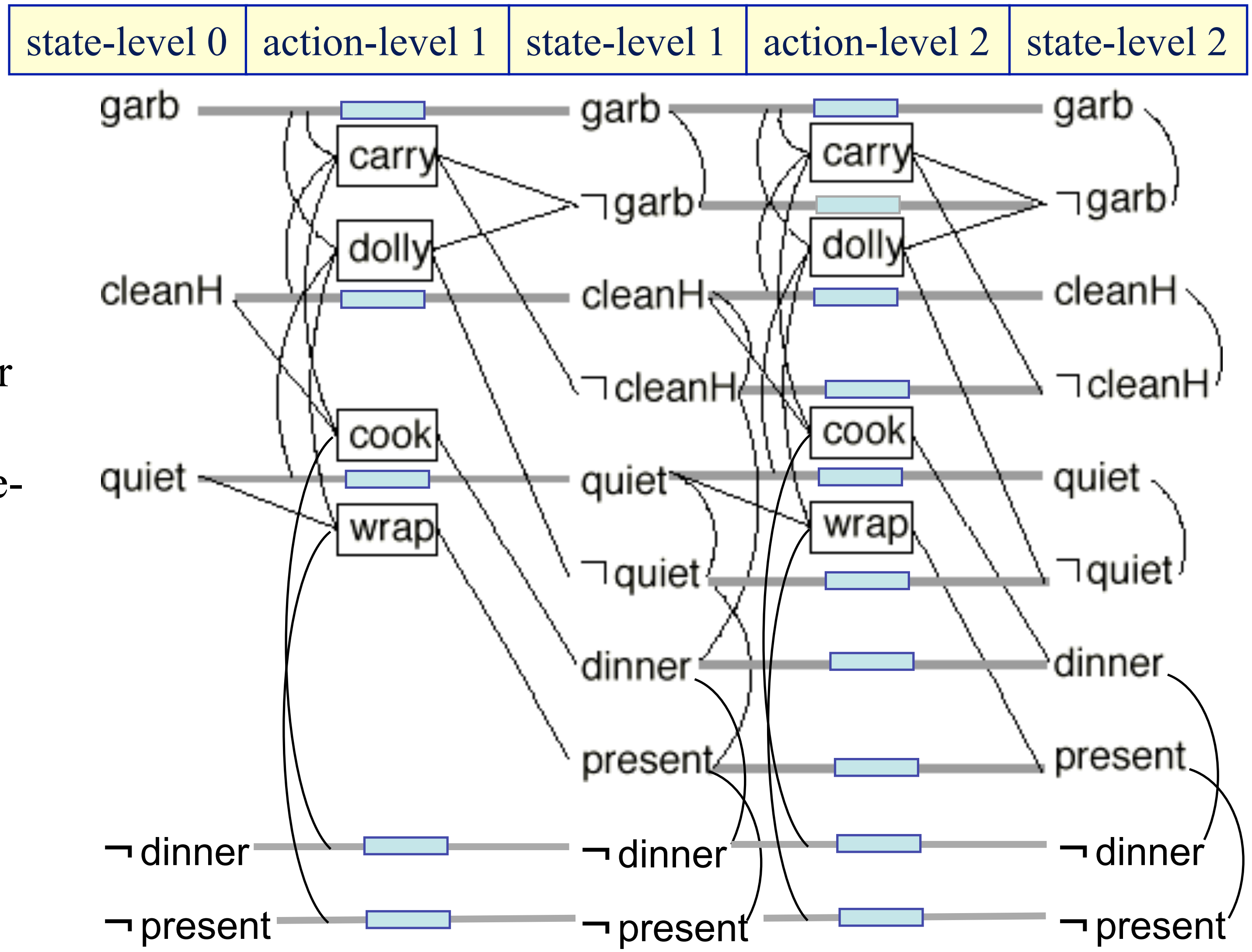
Recall what the algorithm does

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

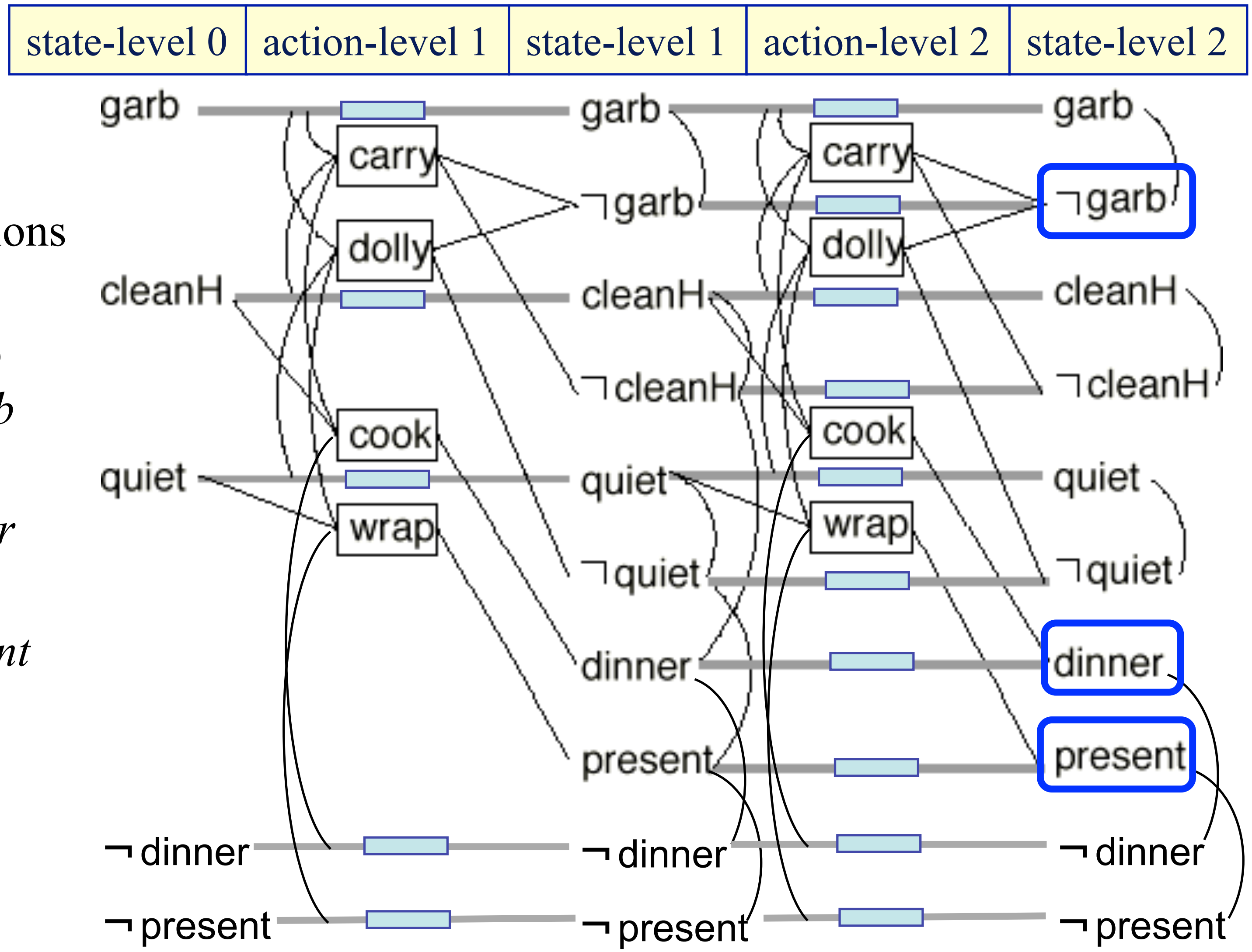
Example (continued)

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



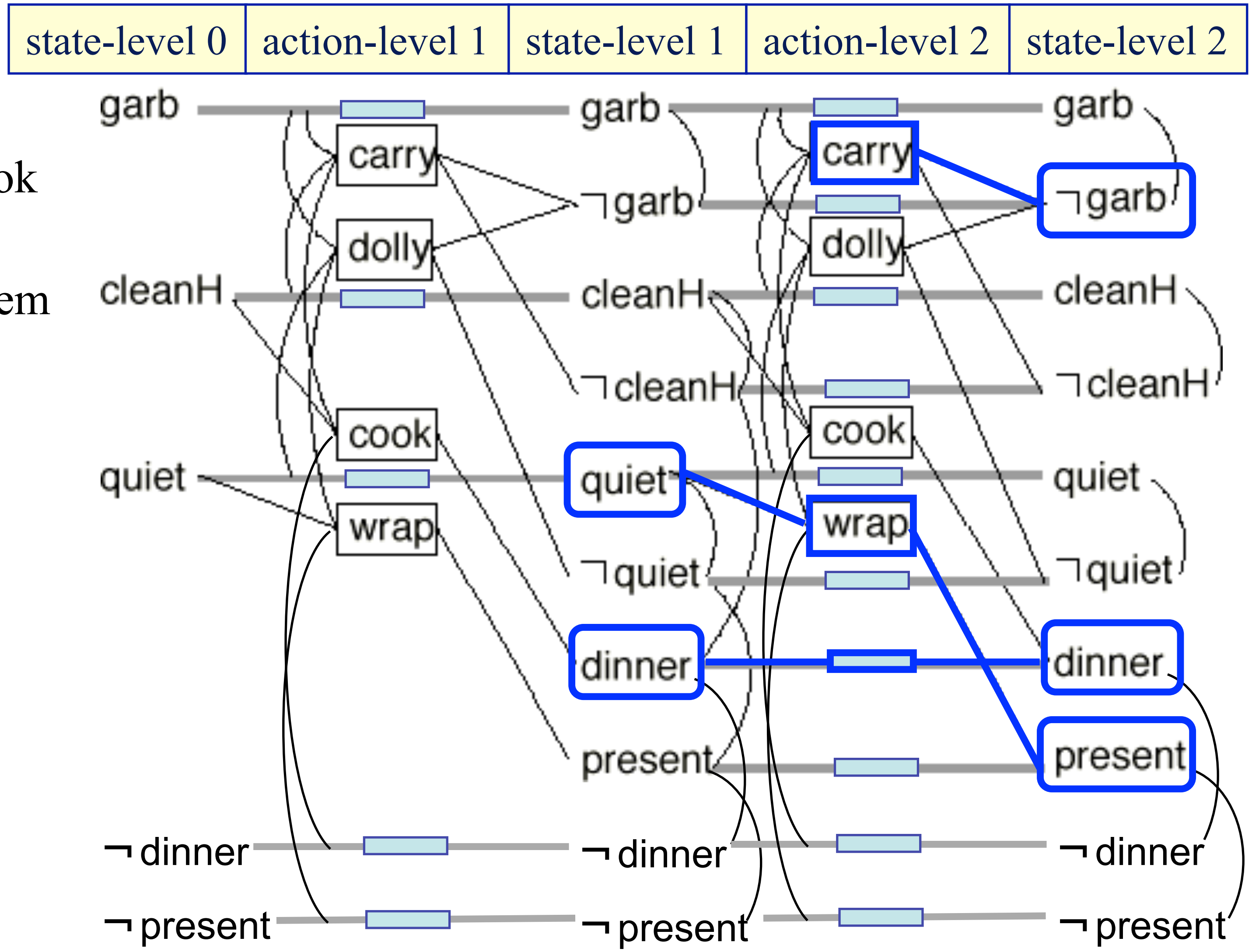
Example (continued)

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



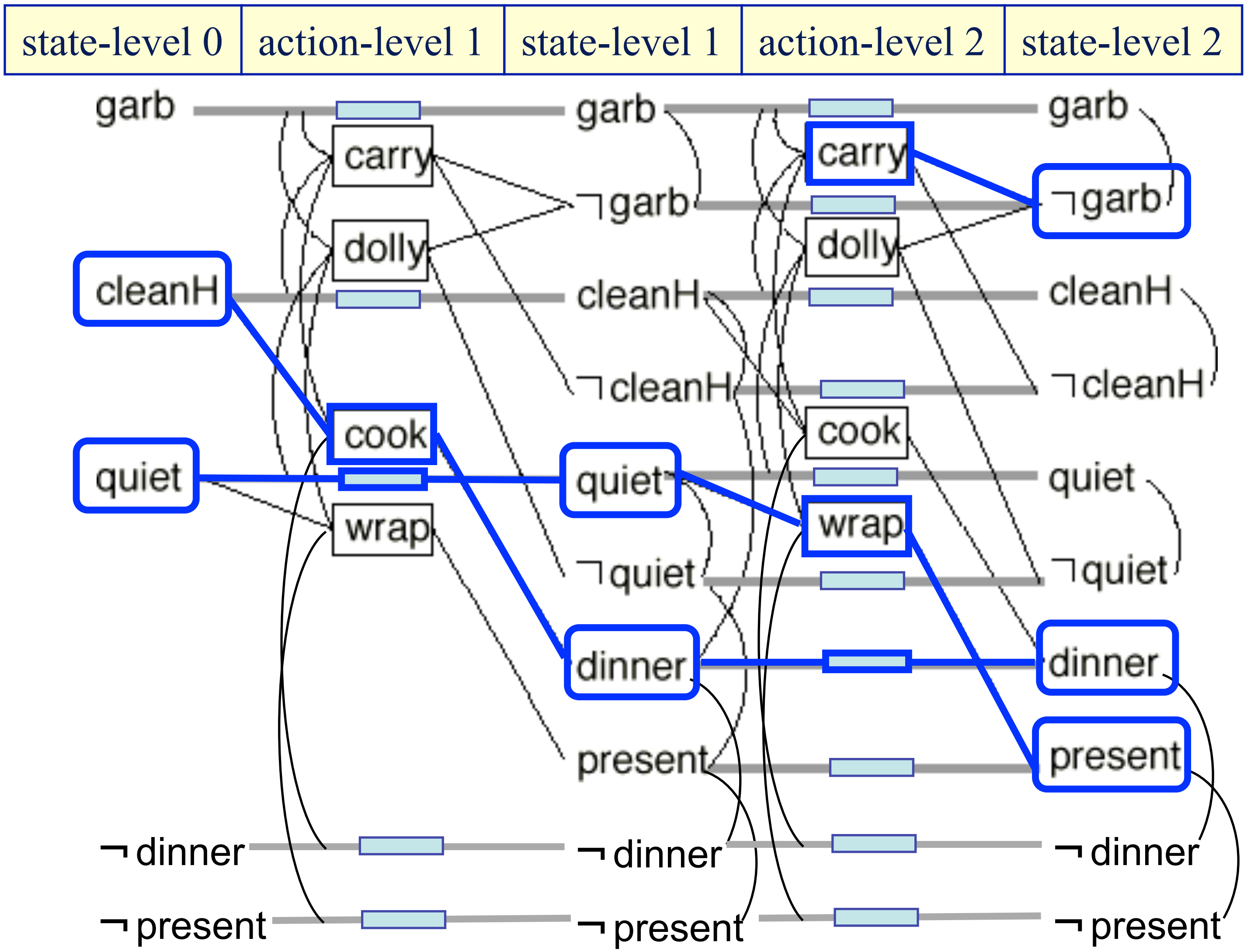
Example (continued)

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

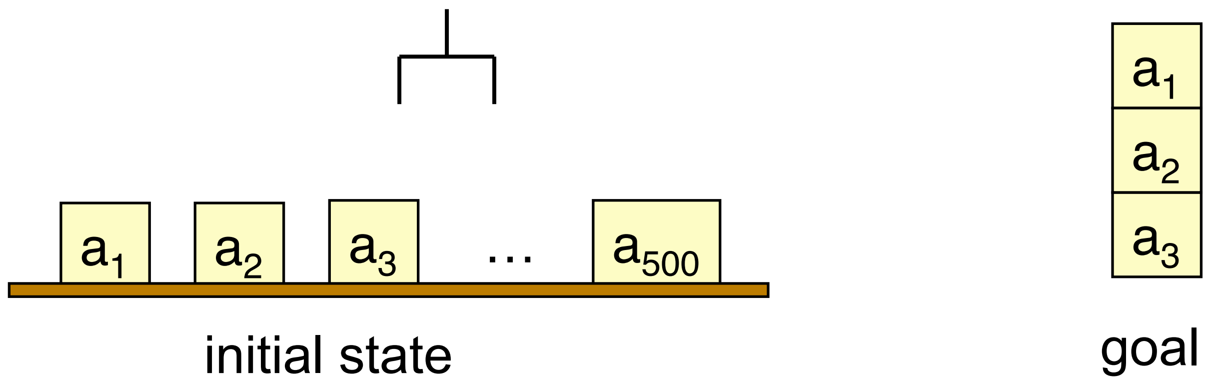


Example (continued)

- 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 have a very large branching factor
    - » pickup(a_1), pickup(a_2) , …, pickup(a_{500})
  - Thus forward-search can waste time trying lots of irrelevant actions
    - » Need a 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

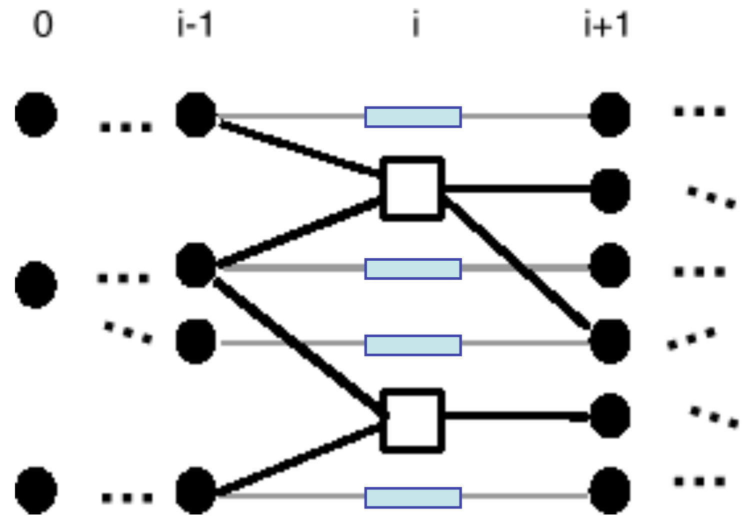
search backward from the goal, looking for a correct plan

if we find one, then return it

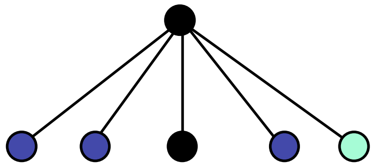
repeat

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



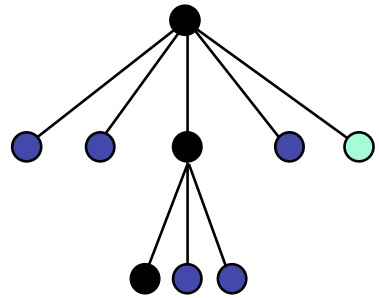
The FastForward Planner



- 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

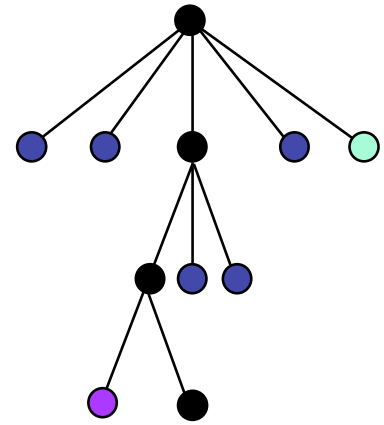


- Use a heuristic function $h(s)$ similar to $\Delta^g(s,g)$
- Don't want an A*-style search (takes too much memory)
- Instead, use a greedy procedure:

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)

The FastForward Planner

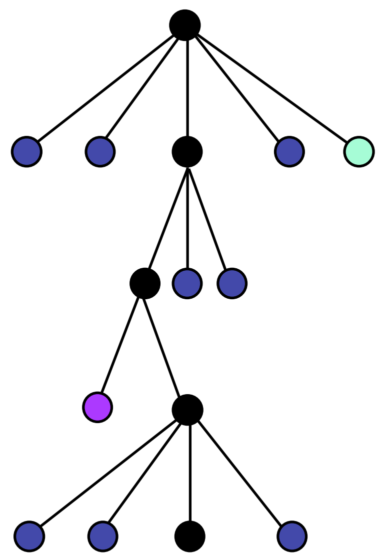
- Use a heuristic function $h(s)$ similar to $\Delta^g(s,g)$
- Don't want an A*-style search (takes too much memory)
- Instead, use a greedy procedure:



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)

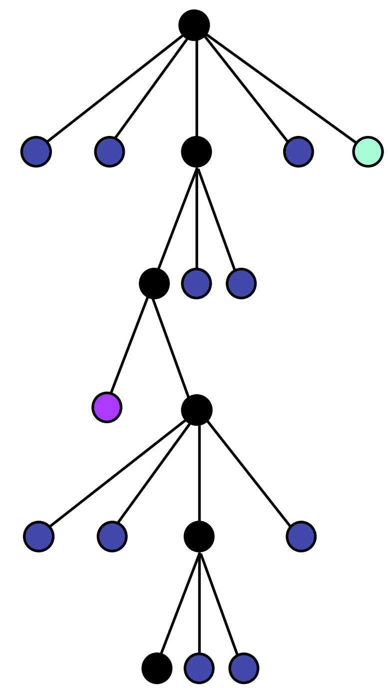
The FastForward Planner

- 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)

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 an “outstanding performance” award
  - Large variance in how good its plans were, but it found them very quickly

Three Main Types of Planners

1. Domain-specific
2. Domain-independent
3. **Configurable**
   - » Domain-independent planning engine
   - » The input includes information about how to plan efficiently in a given problem domain

- I'll now talk about a particular kind of configurable planner

Motivation

- For some planning problems, we may already have ideas about good ways to solve them
- Example: travel to a destination that's far away:
  - Domain-independent planner:
    - » many combinations vehicles and routes
  - Experienced human: small number of "recipes"

    e.g., flying:
    1. buy ticket from local airport to remote airport
    2. travel to local airport
    3. fly to remote airport
    4. travel to final destination
- How to get planning systems to use such recipes?
  - General approach: Hierarchical Task Network (HTN) planning
  - We'll look at a simpler special case: *Task-List Planning*

Task-List Planning

- States and operators: same as in classical planning
- Instead of achieving a *goal*, we will want to accomplish a list of *tasks*
  - Recursively decompose tasks into smaller and smaller subtasks
  - At the bottom, actions that we know how to accomplish directly

- *Task*: an expression of the form $t(u_1,\ldots,u_n)$
  - t is a *task symbol*, and each u_i is a term

- Two kinds of task symbols (and tasks):
  - *primitive*: tasks that we know how to execute directly
    - » task symbol is the head of an operator
  - *nonprimitive*: tasks that must be decomposed into subtasks
    - » use *methods* (next slide)

Methods

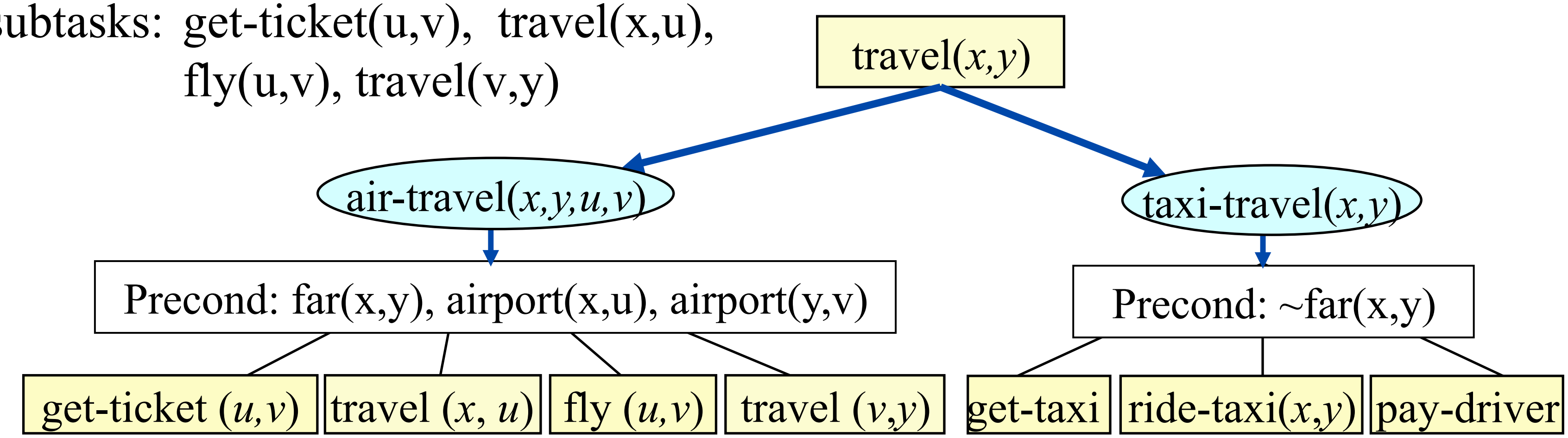
- Method: a 4-tuple $m = (head, task, precond, subtasks)$
  - *head*: the method's *name*, followed by list of variable symbols $(x_1,\ldots,x_n)$
  - *task*: a nonprimitive task
  - *precond*: preconditions (literals)
  - *subtasks*: a sequence of tasks $\langle t_1, \ldots, t_k \rangle$

air-travel(x,y,u,v)

task: travel(x,y)

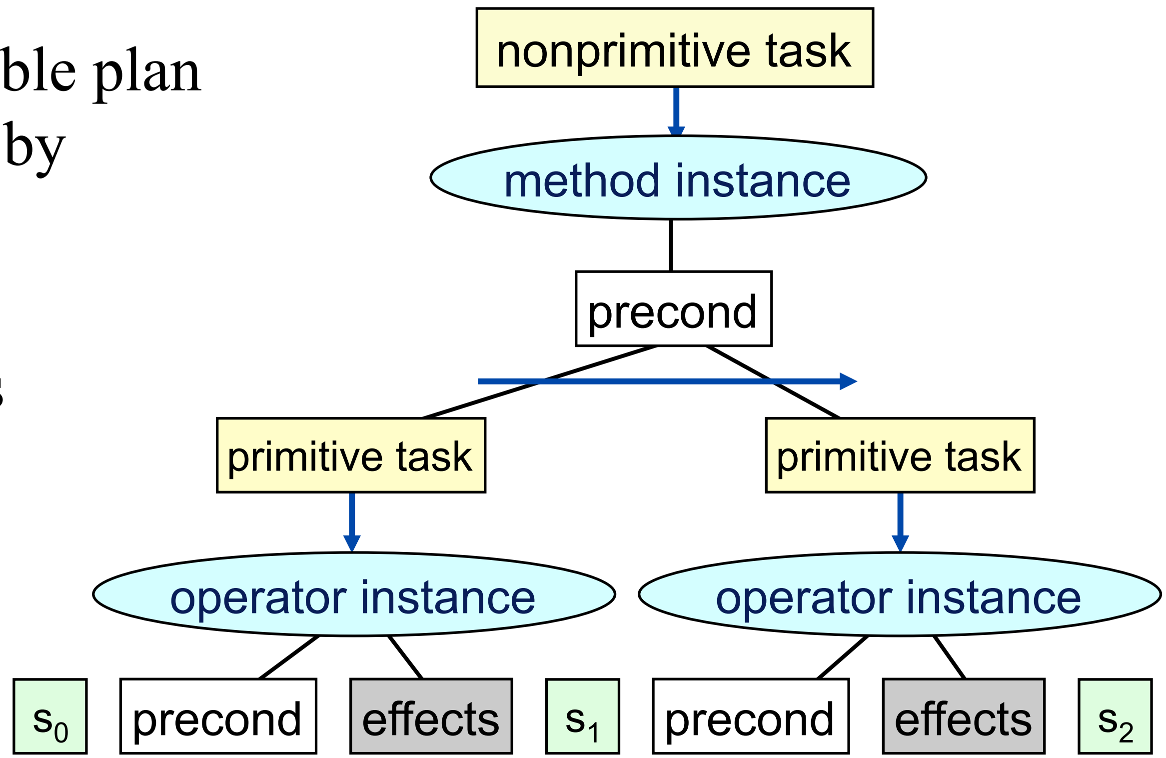
precond: far(x,y), airport(x,u), airport(y,v)

subtasks: get-ticket(u,v), travel(x,u), fly(u,v), travel(v,y)



Domains, Problems, Solutions

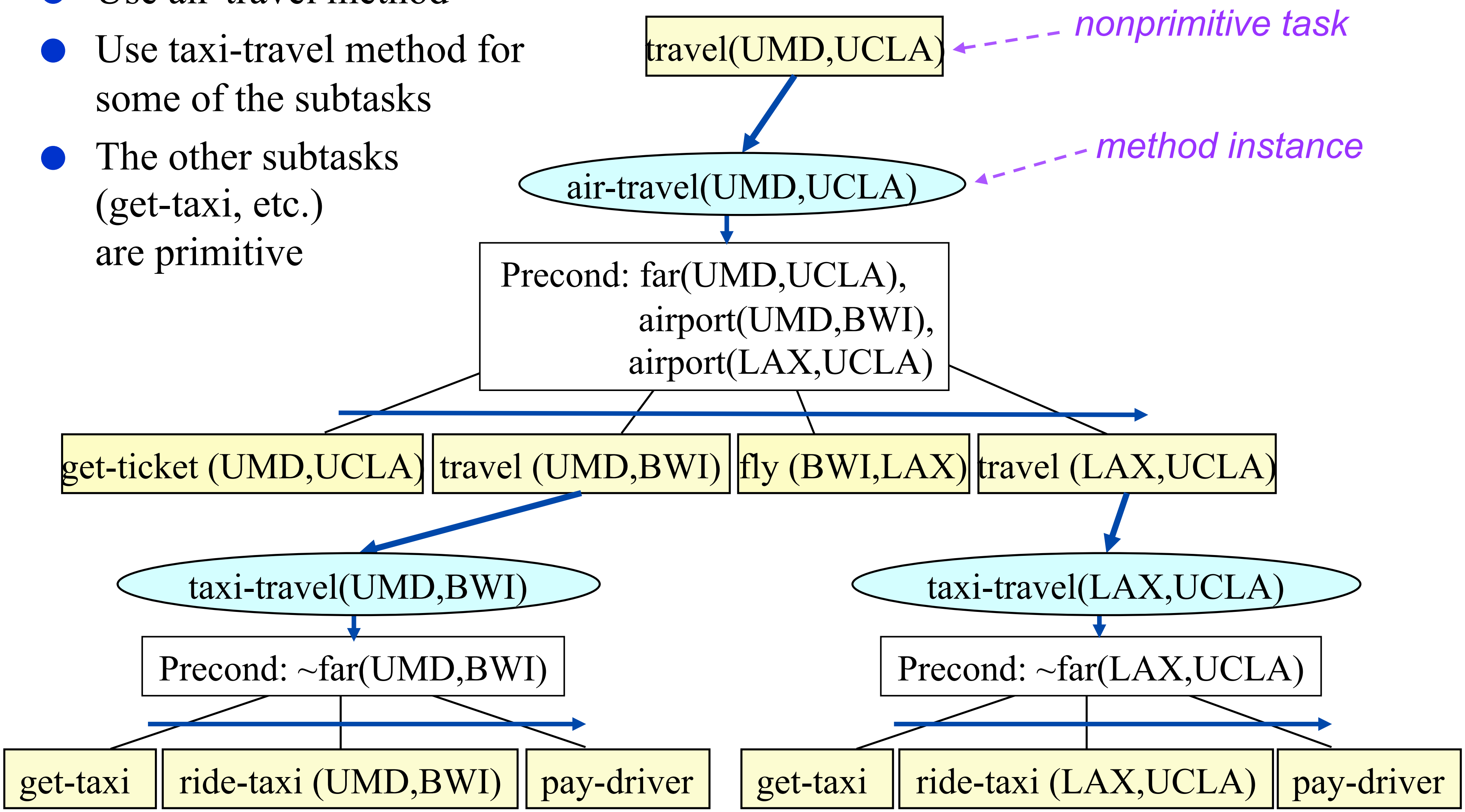
- Task-list planning domain: methods, operators
- Task-list planning problem: methods, operators, initial state, initial task list
- Solution: any executable plan that can be generated by recursively applying
  - methods to nonprimitive tasks
  - operators to primitive tasks



Example

Task: travel from UMD to UCLA

- Use air-travel method
- Use taxi-travel method for some of the subtasks
- The other subtasks (get-taxi, etc.) are primitive



Solving Task-List Planning Problems

- TFD(s,(t_1,…,t_k))
  - if k=0 (i.e., no tasks) then return the empty plan
  - else if there is an action a such that head(a) = t_1 then
    - » if s satisfies precond(a) then
      - return TFD(γ(s,t_1),(t_2,…,t_k))
    - » else return failure
  - else
    - » A = {m : m is a method instance such that task(m)=t_1, and s satisfies precond(m)}
    - » if *active* is empty then return failure
    - » nondeterministically choose m in A
    - » let u_1…, u_j be m's subtasks
    - » return TFD(s, (u_1…, u_j, t_2, …, t_k))

state s; task list T=($\boxed{\mathbf{t_1}}$,t_2,…)
action a

state $\boxed{\gamma(s,a)}$; task list T=(t_2, …)

task list T=($\boxed{\mathbf{t_1}}$,t_2,…)
method instance m

task list T=($\boxed{\mathbf{u_1,\ldots,u_j}}$,t_2,…)

Example

- TFD(s,(t_1,…,t_k))
  - if k=0 (i.e., no tasks) then return the empty plan
  - else if there is an action a such that head(a) = t_1 then
    - » if s satisfies precond(a) then
      - return TFD($\gamma(s,t_1)$,(t_2,…,t_k))
    - » else return failure
  - else
    - » A = {m : m is a method instance such that task(m)=t_1, and s satisfies precond(m)}
    - » if *active* is empty then return failure
    - » nondeterministically choose m in A
    - » let u_1…, u_j be m's subtasks
    - » return TFD(s, (u_1…, u_j, t_2, …, t_k))

s_0: far(UMD,UCLA), airport(UMD,BWI), airport(UCLA,LAX)

task list: ⟨travel(UMD,UCLA)⟩

apply air-travel method: ⟨get-ticket (UMD,UCLA) travel (UMD,BWI) fly (BWI,LAX) travel (LAX,UCLA)⟩

apply get-ticket action: far(UMD,UCLA), airport(UMD,BWI), airport(UCLA,LAX) ticket(UCLA,LAX)

apply taxi-travel method: ⟨get-taxi ride-taxi(UMD,BWI) pay-driver fly (BWI,LAX) travel (LAX,UCLA)⟩

air-travel(x,y,u,v)

Precond: far(x,y), airport(x,u), airport(y,v)

get-ticket (u,v) | travel (x, u) | fly (u,v) | travel (v,y)

taxi-travel(x,y)

Precond: ~far(x,y)

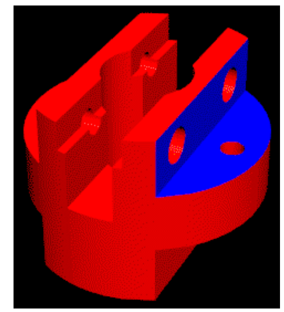
get-taxi | ride-taxi(x,y) | pay-driver

Increasing Expressivity

- Easy to generalize this beyond classical planning
  - States can be arbitrary data structures

Us:	East declarer, West dummy
Opponents:	defenders, South & North
Contract:	East – 3NT
On lead:	West at trick 3

East:	♠KJ74
West:	♠A2
Out:	♠QT98653



  - Preconditions and effects can include
    - » logical inferences (e.g., Horn clauses)
    - » complex numeric computations
    - » interactions with other software packages
- e.g., SHOP and SHOP2

`http://www.cs.umd.edu/projects/shop`

Example

method travel-by-foot(a,x,y)
  precond: $distance(x,y) \leq 2$
  task: travel(a,x,y)
  subtasks: walk(a,x,y)

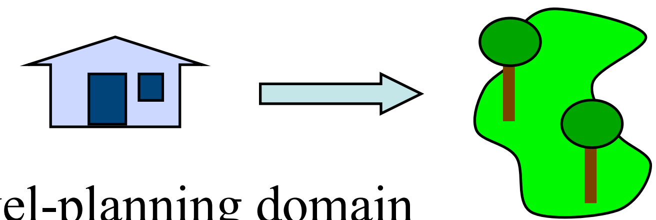
method travel-by-taxi(a,x,y)
  task: travel(a,x,y)
  precond: $cash(a) \geq 1.5 + 0.5 \times distance(x,y)$
  subtasks: $\langle$call-taxi(a,x), ride(a,x,y), pay-driver$(a,x,y)\rangle$

operator walk(a,x,y)
  precond: $location(a) = x$
  effects: $location(a) \leftarrow y$

operator call-taxi(a,x)
  effects: $location(taxi) \leftarrow x$

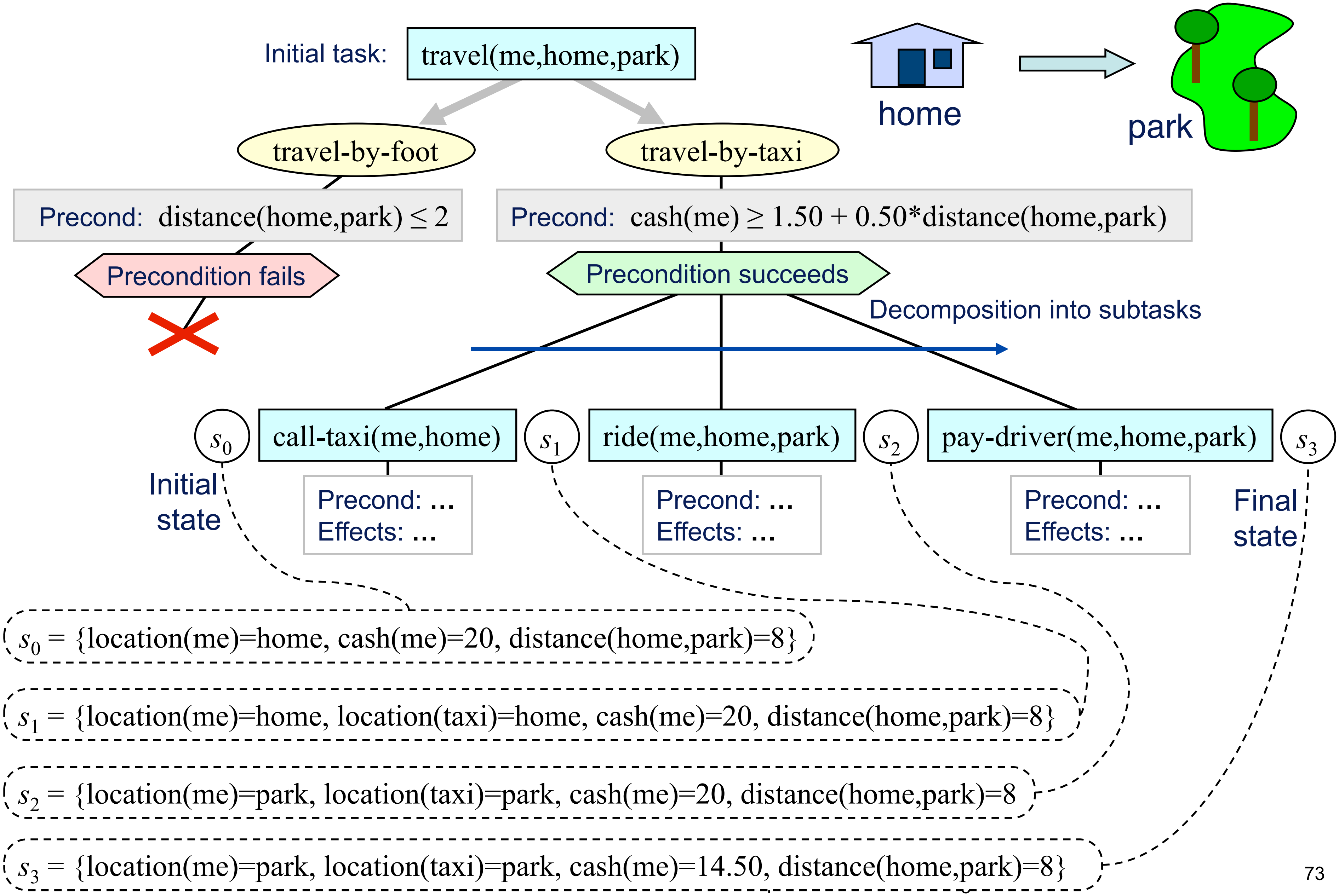
operator ride-taxi(a,x,y)
  precond: $location(taxi) = x$, $location(a) = x$
  effects: $location(taxi) \leftarrow y$, $location(a) \leftarrow y$

operator pay-driver(a,x,y)
  precond: $cash(a) \geq 1.5 + 0.5 \times distance(x,y)$
  effects: $cash(a) \leftarrow cash(a) - 1.5 + 0.5 \times distance(x,y)$



- Simple travel-planning domain
  - Go from one location to another
  - State = {values of variables}

Planning Problem: I am at home, I have $20, I want to go to a park 8 miles away



Comparison to Classical Planners

- Advantages:
  - Can encode “recipes” (standard ways do planning in a given domain) as collections of methods and operators
    - » Helps the planning system do more-intelligent search - can speed up planning by many orders of magnitude (e.g., polynomial time versus exponential time)
    - » Produces plans that correspond to how a human might solve the problem
  - Greater expressive power
    - » Preconditions and effects can be computational algorithms
- Disadvantages:
  - More complicated than just writing classical operators
  - The author needs knowledge about planning in the given domain

SHOP2

- SHOP2:
  - `http://www.cs.umd.edu/projects/shop`
  - Algorithm is a generalized version of TFD
  - Won an award in the AIPS-2002 Planning Competition
  - Freeware, open source
  - Downloaded more than 13,000 times
  - Used in hundreds (thousands?) of projects worldwide