Part 2

—

KRR for Dynamic Environments

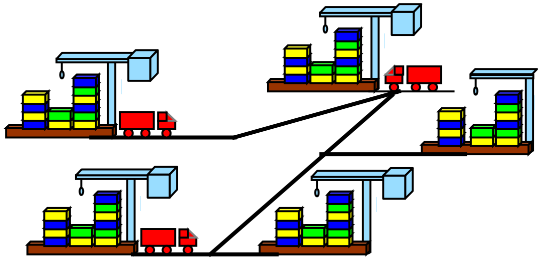
(Week 6 – 9)

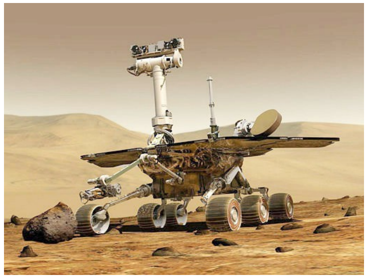
Dynamic Environments

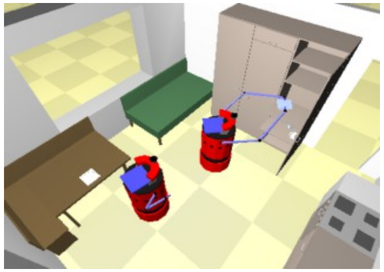
Part 2 is concerned with KRR techniques for

- intelligent (virtual) agents
- intelligent robos
- …

that interact with an evironment







Overview: Representation Techniques

Week 6

- Representations for classical planning problems
  - deterministic environment; complete information

Week 7

- Logic programs for problem representations
  - including planning problems, games

Week 8

- First-order logic to describe dynamic environments
  - deterministic environment; (in-)complete information

Week 9

- State transition systems to describe dynamic environments
  - nondeterministic environment; (in-)complete information

Overview: Reasoning Techniques

Week 6
- Planning algorithms

Week 7
- Answer Set Programming

Week 8
- Reasoning in first-order logic, Prolog

Week 9
- Markov decision algorithms
- Decision making

Planning

- Representations for classical planning
- Modern heuristics for state-space planning
- Planning graphs: a modern planning technique

Background reading

Automated Planning by Malik Ghallab, Dana Nau, Paolo Traverso, Morgan Kaufmann 2004. Chapters 1, 2, 4 & 6

Some Dictionary Definitions of “Plan”

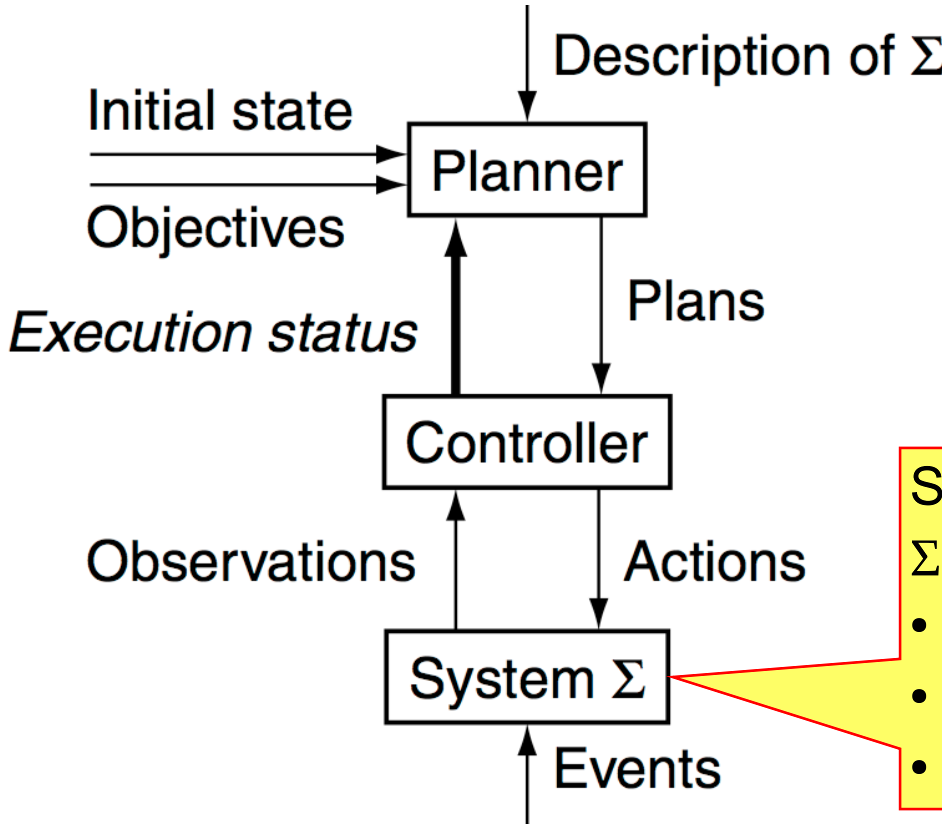
plan *n.*

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.*

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

– Austin Tate, *MIT Encyclopaedia of the Cognitive Sciences*, 1999

Planning for an Agent/Robot in a Dynamic World

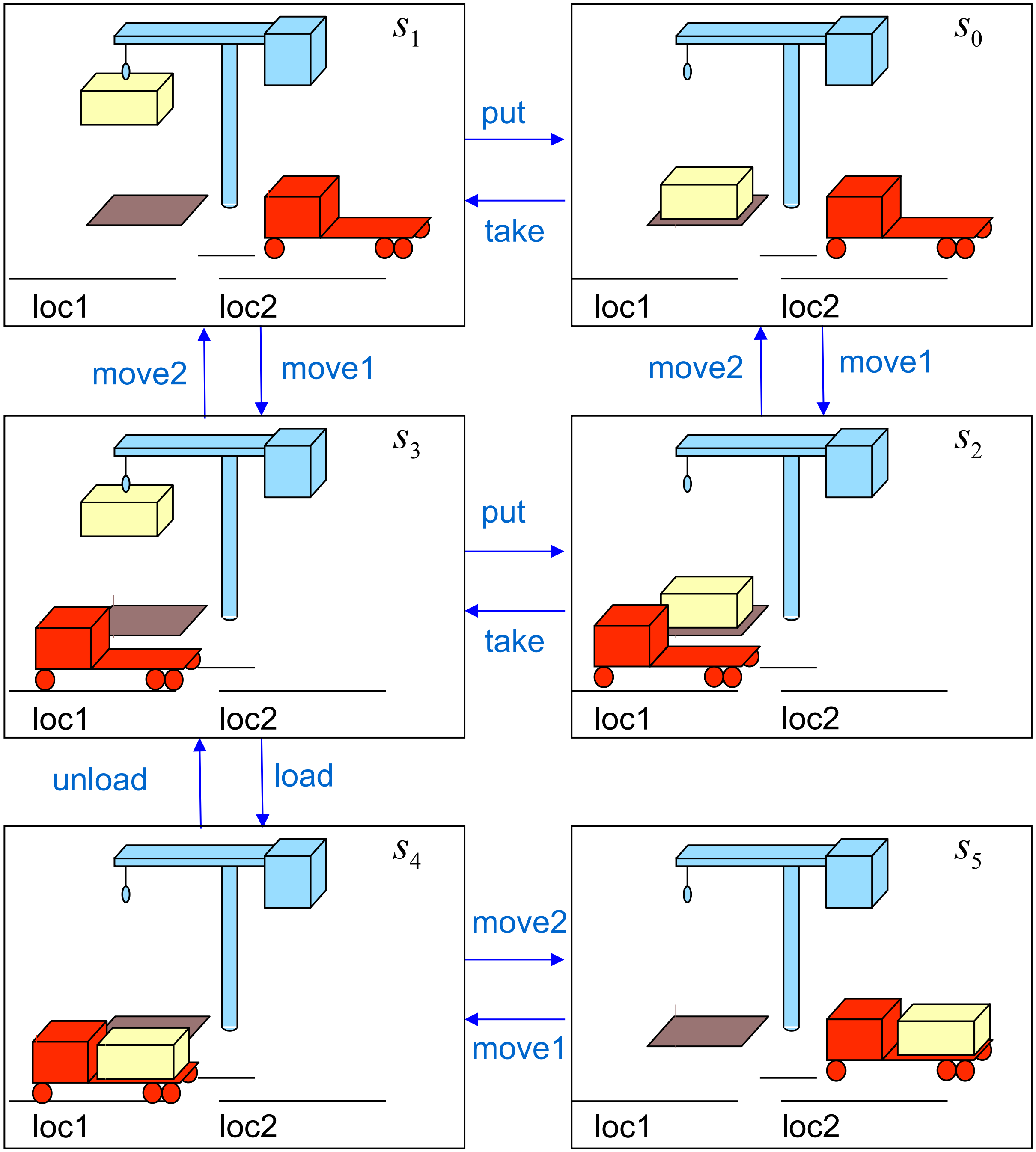


State transition system
$\Sigma = (S,A,\gamma)$
- S = {states}
- A = {actions}
- γ = state-transition function

- Σ is an abstraction that deals only with the aspects that the planner needs to reason about

Example

- Example $\Sigma = (S,A,\gamma)$:
  - $S = \{s_0, \ldots, s_5\}$
  - A = {move1, move2, put, take, load, unload}
  - γ: see the arrows

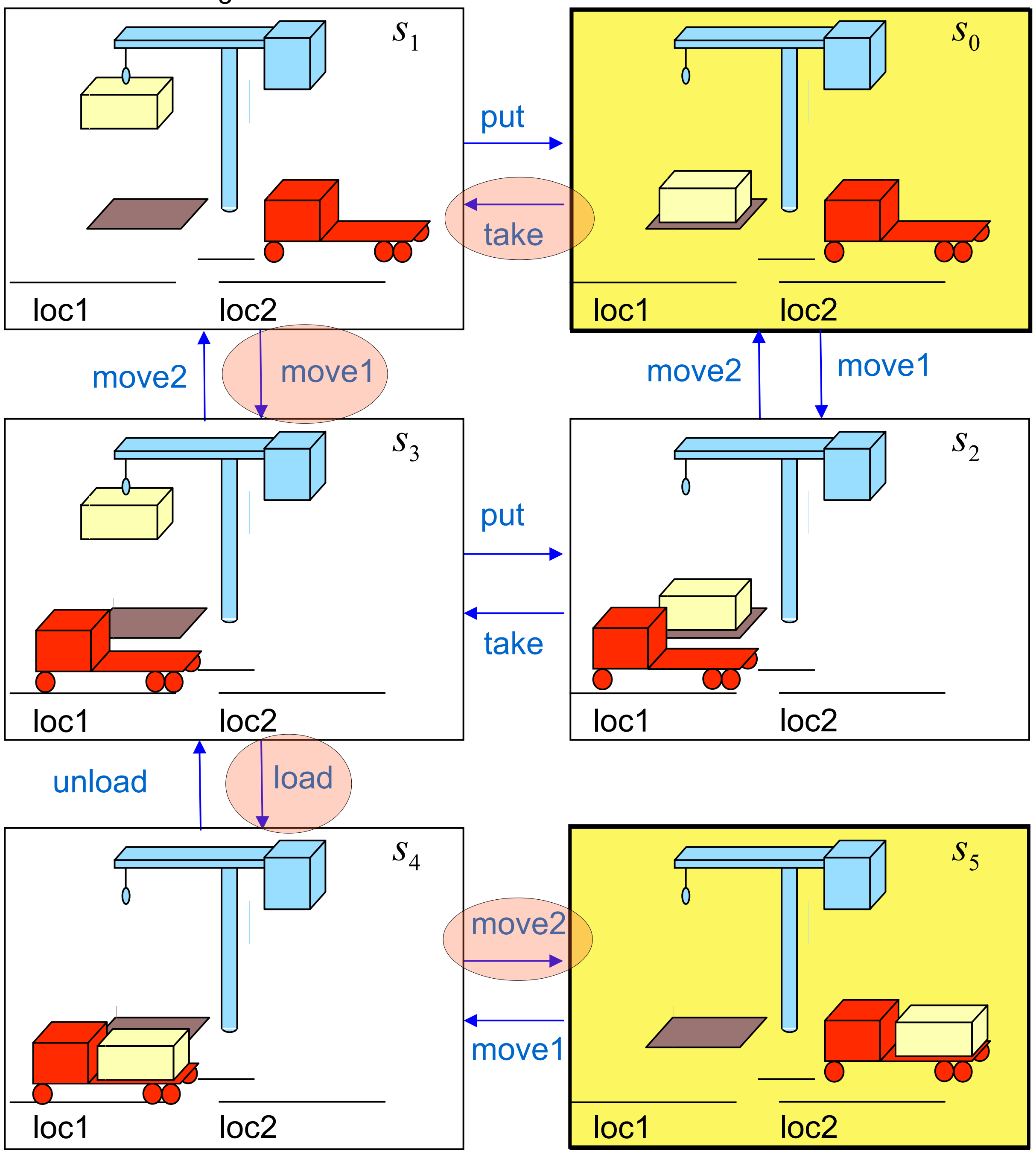


Dock Worker Robots (DWR) example

Example

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

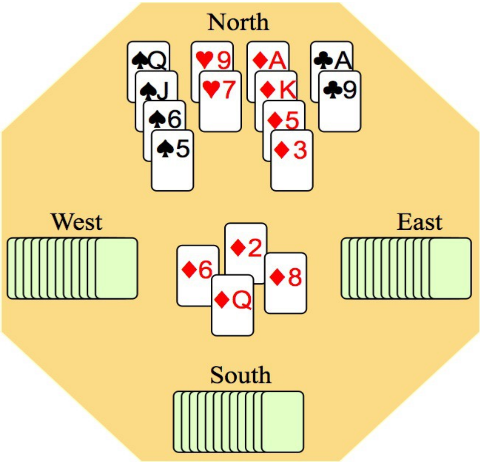
  ⟨take, move1, load, move2⟩

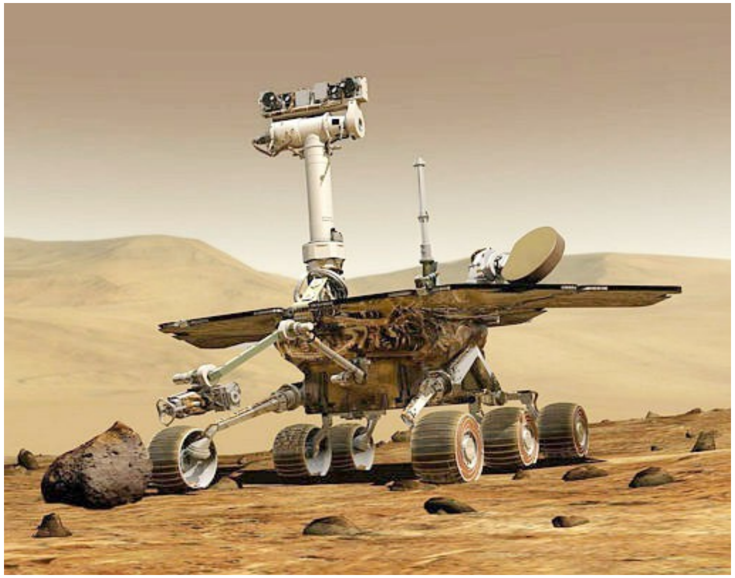


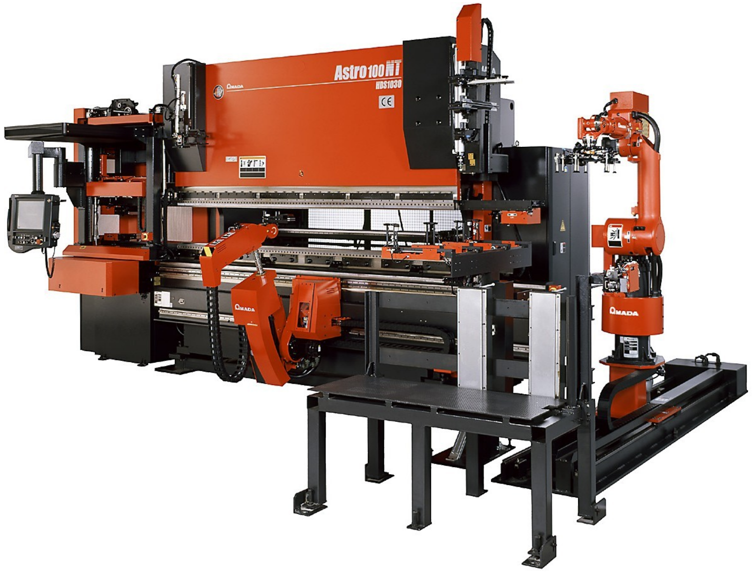
Dock Worker Robots (DWR) example

Domain-Specific Planners

- Most successful real-world planning systems work this way
  - Mars exploration, sheet-metal bending, playing bridge, etc.
- Often use problem-specific techniques that are difficult to generalise to other planning domains

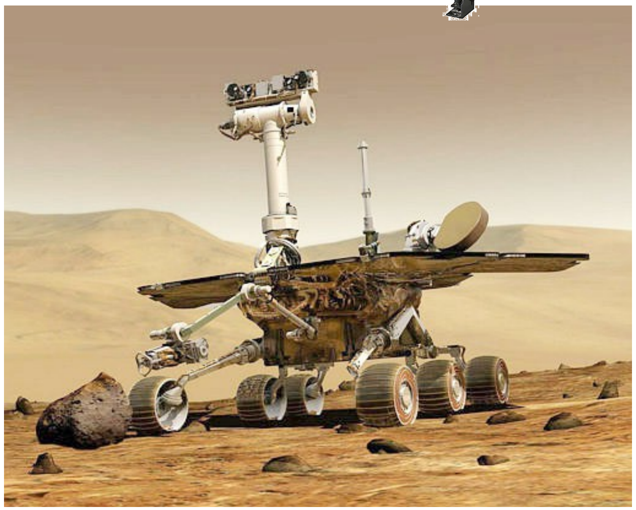






Domain-Independent Planners

- No domain-specific knowledge except the description of the system Σ
- In practice,
  - Not feasible to make domain-independent planners work well in all possible planning domains



- Make simplifying assumptions to restrict the set of domains
  - **Classical planning**

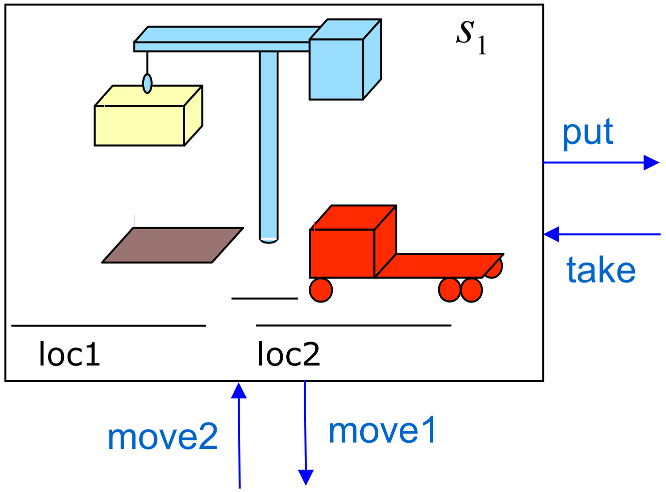
    ⇨ Historical focus of most research on automated planning

Classical Planning

- Reduces to the following problem:
  Given Σ, initial state s_0, and goal states S_g,
  find a sequence of actions $(a_1, a_2, \ldots a_n)$ that produces
  a sequence of state transitions $(s_0, s_1, s_2, \ldots, s_n)$ such that $s_n \in S_g$

Is this trivial?

- Generalise the earlier example:
  - Five locations, three robot carts, 100 containers, three piles
    ⇨ 10^{277} states



- Automated-planning research has been heavily dominated by classical planning. There are dozens of different algorithms.

Representations for Classical Planning

Classical Representations: Motivation

- In most problems, far too many states to try to represent all of them explicitly as s_0, s_1, s_2, …

  ⇨ represent each state as a set of **atomic features**

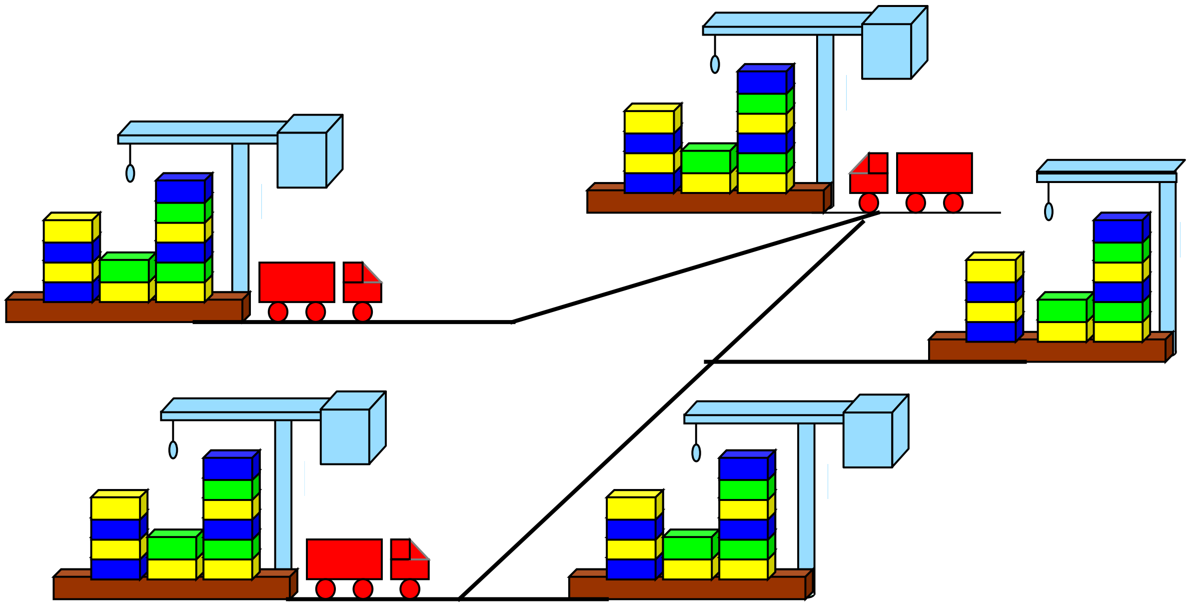
- Define a set of **operators** that can be used to compute state-transitions
- Don't give all of the states explicitly
  - Just give the initial state
  - Use the operators to generate the other states as needed

Classical Representation

- Language of first-order logic but without function symbols

  ⇨ finitely many predicate symbols and constant symbols

- Example: the DWR domain
  - Locations: l1, l2, …
  - Containers: c1, c2, …
  - Piles: p1, p2, …
  - Robot carts: r1, r2, …
  - Cranes: k1, k2, …



Example (cont'd)

- **Fixed relations**: same in all states

  adjacent(*l,l'*)   attached(*p,l*)   belong(*k,l*)

- **Dynamic relations**: differ from one state to another

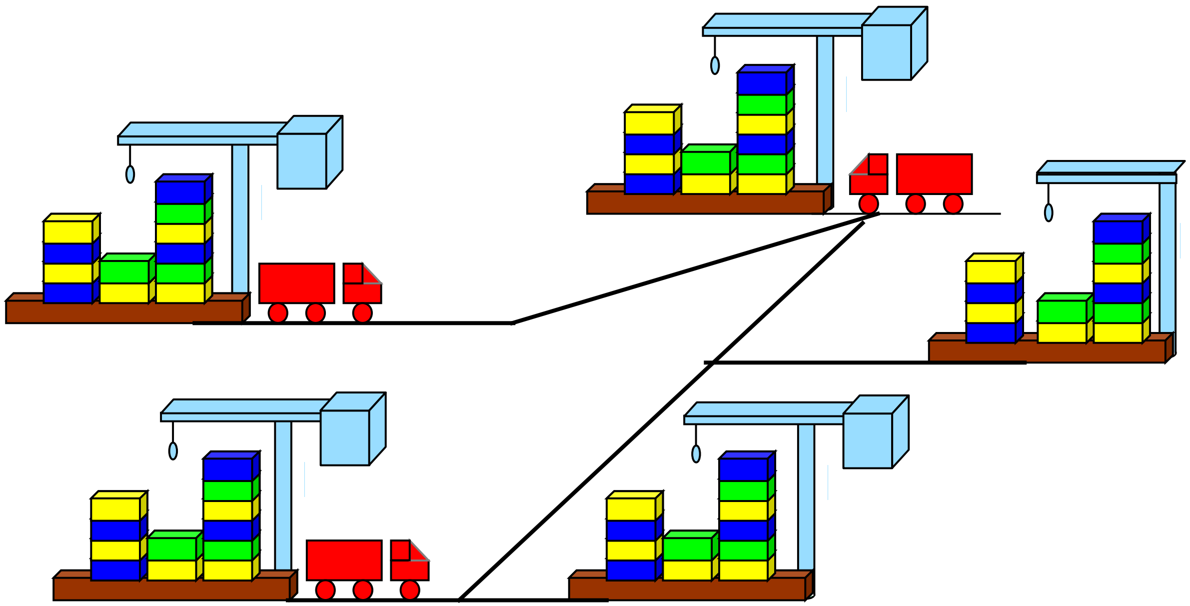
  occupied(*l*)   at(*r,l*)

  loaded(*r,c*)   unloaded(*r*)

  holding(*k,c*)   empty(*k*)

  in(*c,p*)   on(*c,c'*)

  top(*c,p*)   top(pallet,*p*)



- **Actions**:

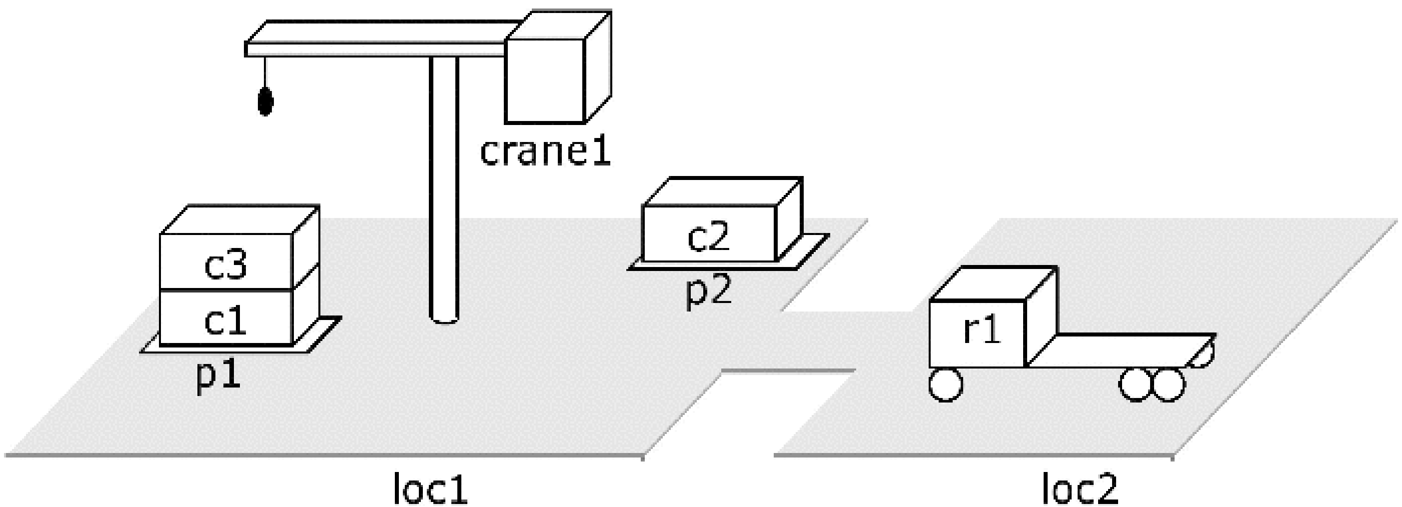
  take(*c,k,p*)   put(*c,k,p*)

  load(*r,c,k*)   unload(*r*)   move(*r,l,l'*)

States

A **state** is a set s of ground atoms

- The atoms represent the things that can be true in some states
- Only finitely many ground atoms, so only finitely many possible states



s_1 = {attached(p1,loc1), in(c1,p1), in(c3,p1), top(c3,p1), on(c3,c1), on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}

Operators

An **operator** is a triple o = (name(o), precond(o), effects(o))

- name(o): a syntactic expression of the form $n(x_1,\ldots,x_k)$
  - $(x_1,\ldots,x_k)$ is a list of every variable symbol (parameter) that appears in o
- precond(o): **preconditions**
  - literals that must be true in order to use the operator
- effects(o): **effects**
  - literals the operator will make true

Example

```
take(k,l,c,d,p)
  ;; crane k at location l takes c off of d in pile p
  precond: belong(k,l), attached(p,l),empty(k), top(c,p),
           on(c,d)
  effects: holding(k,c), ¬empty(k), ¬in(c,p), ¬top(c,p),
           ¬on(c,d), top(d,p)
```

Actions

An **action** is a ground instance (via a substitution) of an operator

```
take(k,l,c,d,p)
  ;; crane k at location l takes c off of d in pile p
  precond: belong(k,l), attached(p,l),empty(k), top(c,p),
           on(c,d)
  effects: holding(k,c), ¬empty(k), ¬in(c,p), ¬top(c,p),
           ¬on(c,d), top(d,p)
```

- Let σ = {k / crane1, l / loc1, c / c3, d / c1, p / p1}
- Then take(k,l,c,d,p)σ is the following action:

  take(crane1,loc1,c3,c1,p1)

  precond: belong(crane1,loc1), attached(p1,loc1), empty(crane1), top(c3,p1), on(c3,c1)

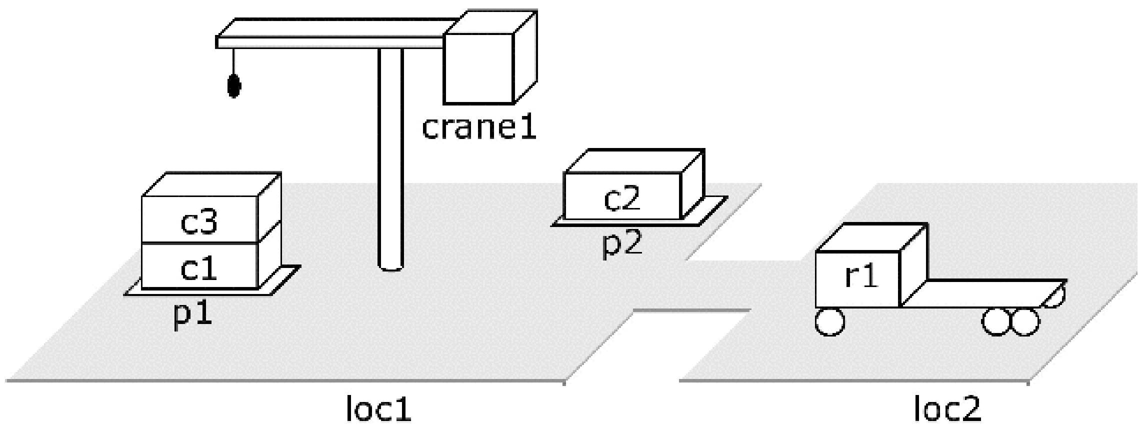
  effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)

Applicability and Result of Actions

- 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 a's preconditions}
  - $\text{precond}^-(a)$ = {atoms that appear negatively in a's preconditions}
  - $\text{effects}^+(a)$ = {atoms that appear positively in a's effects}
  - $\text{effects}^-(a)$ = {atoms that appear negatively in a's effects}

- Action a is **applicable** to (or **executable** in) S if
  - $\text{precond}^+(a) \subseteq s$
  - $\text{precond}^-(a) \cap s = \emptyset$
- The **result** of applying action a to state S is
  - $\gamma(s,a) = (s \setminus \text{effects}^-(a)) \cup \text{effects}^+(a)$

Example: Applicability



- An action:

  take(crane1,loc1,c3,c1,p1)

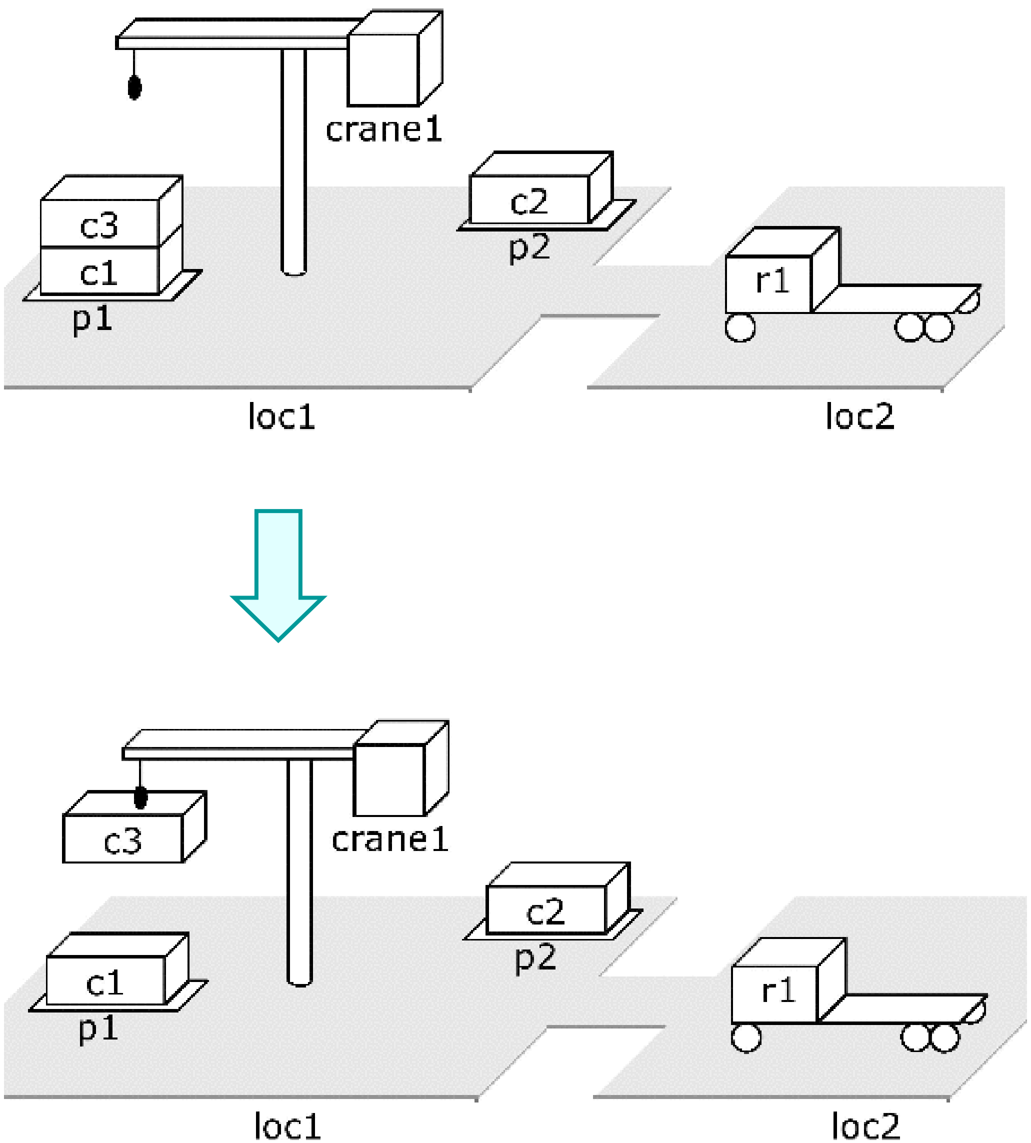
  precond: belong(crane,loc1),
  attached(p1,loc1),
  empty(crane1), top(c3,p1),
  on(c3,c1)

  effects: holding(crane1,c3),
  ¬empty(crane1),
  ¬in(c3,p1), ¬top(c3,p1),
  ¬on(c3,c1), top(c1,p1)

- A state it's applicable to

  s_1 = {**attached(p1,loc1)**, in(c1,p1), in(c3,p1),
  top(c3,p1), **on(c3,c1)**, on(c1,pallet),
  attached(p2,loc1), in(c2,p2),
  top(c2,p2), on(c2,pallet),
  belong(crane1,loc1), **empty(crane1)**,
  adjacent(loc1,loc2),
  adjacent(loc2,loc1), at(r1,loc2),
  occupied(loc2, unloaded(r1)}

Example: Result



take(crane1,loc1,c3,c1,p1)

precond: belong(crane,loc1),
attached(p1,loc1),
empty(crane1), top(c3,p1),
on(c3,c1)

effects: holding(crane1,c3),
¬empty(crane1),
¬in(c3,p1), ¬top(c3,p1),
¬on(c3,c1), top(c1,p1)

s_2 = {attached(p1,loc1), in(c1,p1), ~~in(c3,p1)~~,
~~top(c3,p1)~~, ~~on(c3,c1)~~, on(c1,pallet),
attached(p2,loc1), in(c2,p2),
top(c2,p2), on(c2,pallet),
belong(crane1,loc1), ~~empty(crane1)~~,
adjacent(loc1,loc2),
adjacent(loc2,loc1), at(r1,loc2),
occupied(loc2, unloaded(r1),
holding(crane1,c3), **top(c1,p1)**}

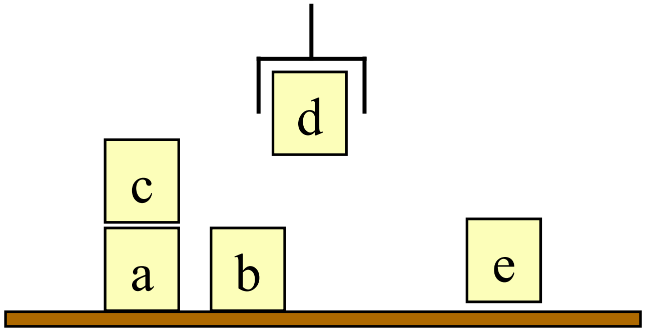
Exercise

Exercise: The Blocks World

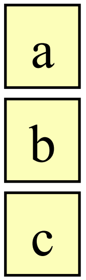
- Infinitely wide table, finite number of children’s blocks
- Ignore where a block is located on the table
- A block can sit on the table or on another block
- There’s a robot gripper that can hold at most one block

- Want to move blocks from one configuration to another
  - e.g.,

initial state

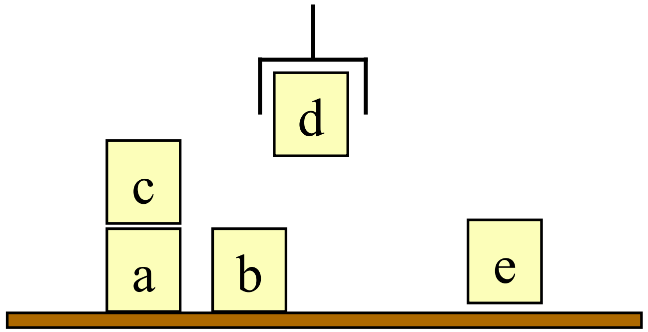


goal



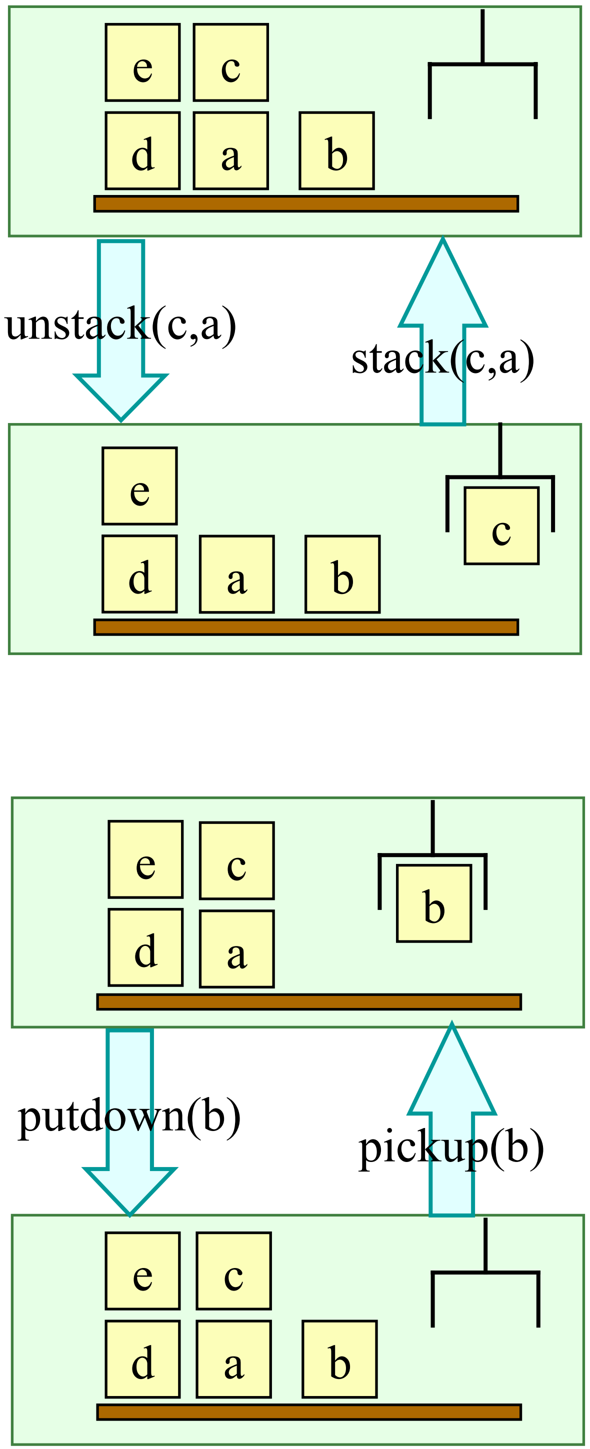
Exercise: Classical Representation – Symbols

- Constant symbols:
  - The blocks: a, b, c, d, e
- Dynamic relations?



Exercise: Classical Operators

- Preconditions and effects?



Summary: Planning Problems

Given a planning domain (language L, operators O)

- **Representation** of a planning problem: a triple $P = (O,s_0,g)$
  - O is the collection of operators
  - s_0 is a state (the initial state)
  - g is a set of literals (the goal formula)

Plans and Solutions

Let $P = (O, s_0, g)$ be a planning problem

- **Plan**: any sequence of actions $\pi = \langle a_1, a_2, \ldots, a_n \rangle$ such that each a_i is an instance of an operator in O
- Plan π is a **solution** for $P = (O, s_0, g)$ if it is executable and achieves g
  - i.e., if there are states $s_0, s_1, \ldots, s_n$ such that

$$\gamma(s_0, a_1) = s_1$$

$$\gamma(s_1, a_2) = s_2$$

$$\vdots$$

$$\gamma(s_{n-1}, a_n) = s_n$$

s_n satisfies g

Example: The 5 DWR Operators

move(r, l, m)
;; robot r moves from location l to location m
precond: adjacent(l, m), at(r, l), $\neg$ occupied(m)
effects: at(r, m), occupied(m), $\neg$ occupied(l), $\neg$ at(r, l)

load(k, l, c, r)
;; crane k at location l loads container c onto robot r
precond: belong(k, l), holding(k, c), at(r, l), unloaded(r)
effects: empty(k), $\neg$ holding(k, c), loaded(r, c), $\neg$ unloaded(r)

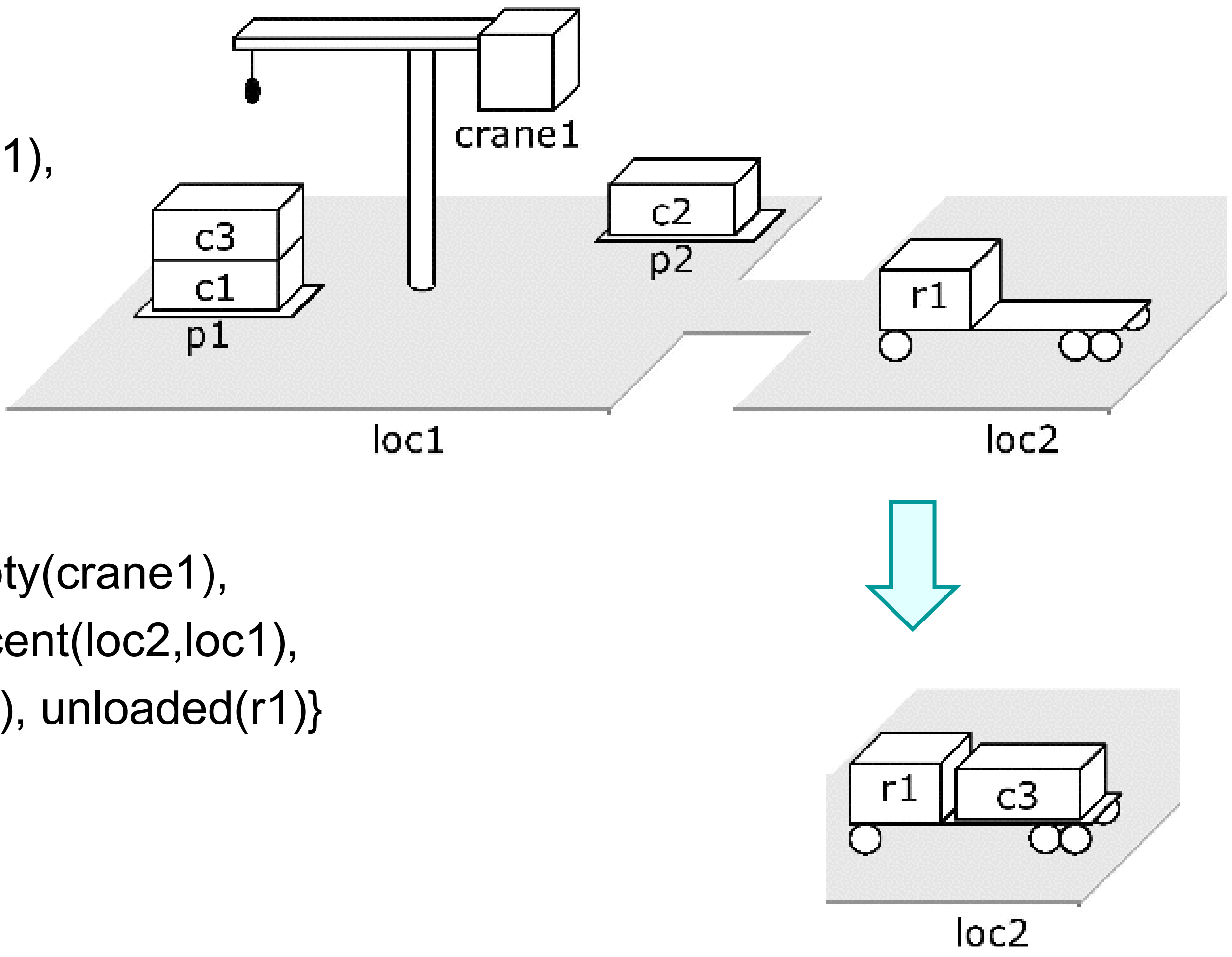
unload(k, l, c, r)
;; crane k at location l takes container c from robot r
precond: belong(k, l), at(r, l), loaded(r, c), empty(k)
effects: $\neg$ empty(k), holding(k, c), unloaded(r), $\neg$ loaded(r, c)

put(k, l, c, d, p)
;; crane k at location l puts c onto d in pile p
precond: belong(k, l), attached(p, l), holding(k, c), top(d, p)
effects: $\neg$ holding(k, c), empty(k), in(c, p), top(c, p), on(c, d), $\neg$ top(d, p)

take(k, l, c, d, p)
;; crane k at location l takes c off of d in pile p
precond: belong(k, l), attached(p, l), empty(k), top(c, p), on(c, d)
effects: holding(k, c), $\neg$ empty(k), $\neg$ in(c, p), $\neg$ top(c, p), $\neg$ on(c, d), top(d, p)

Example

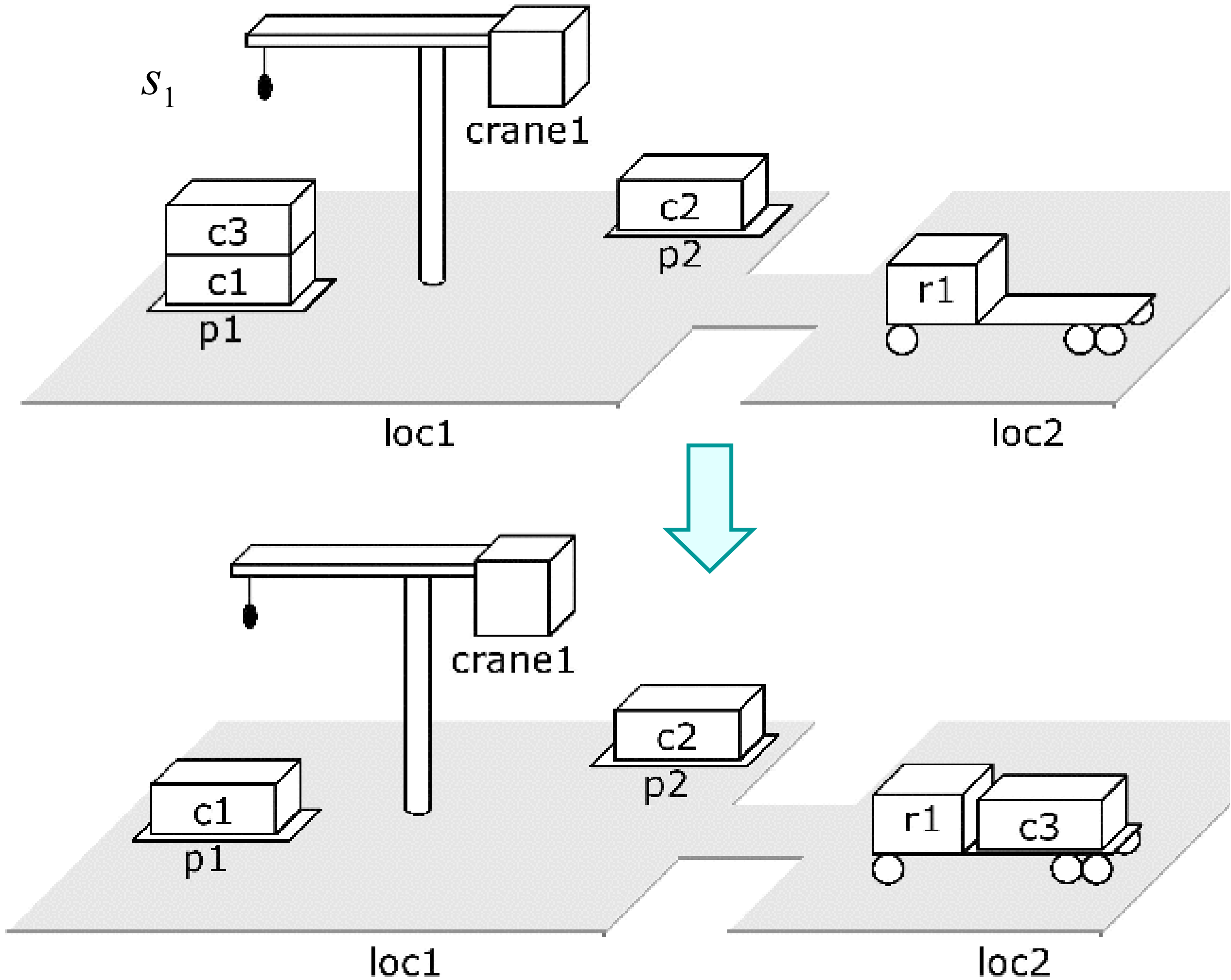
- Let $P = (O, s_0, g)$, where
  - O = {the 5 DWR operators}
  - s_0 = {attached(p1,loc1), in(c1,p1), in(c3,p1), top(c3,p1), on(c3,c1), on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}
  - g = {loaded(r1,c3), at(r1,loc2)}



- Two *redundant* solutions (can remove actions and still have a solution):

  ⟨move(r1,loc2,loc1),
  take(crane1,loc1,c3,c1,p1),
  ~~move(r1,loc1,loc2),~~
  ~~move(r1,loc2,loc1),~~
  load(crane1,loc1,c3,r1),
  move(r1,loc1,loc2)⟩

  ⟨take(crane1,loc1,c3,c1,p1),
  ~~put(crane1,loc1,c3,c2,p2),~~
  move(r1,loc2,loc1),
  ~~take(crane1,loc1,c3,c2,p2),~~
  load(crane1,loc1,c3,r1),
  move(r1,loc1,loc2)⟩



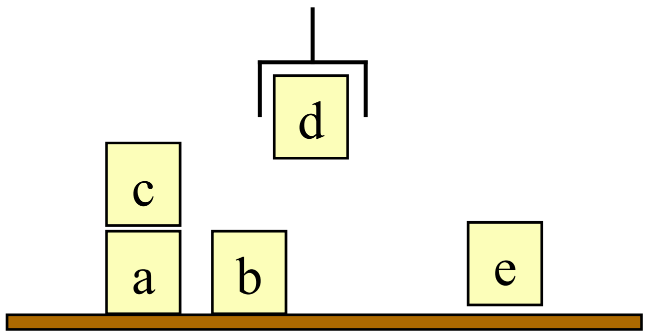
- A solution that is both *irredundant* and *shortest*:
  ⟨move(r1,loc2,loc1), take(crane1,loc1,c3,c1,p1),
  load(crane1,loc1,c3,r1), move(r1,loc1,loc2)⟩

- Are there any other shortest solutions? Are irredundant solutions always shortest?

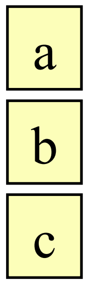
Exercise

Exercise: Plans

initial state



goal

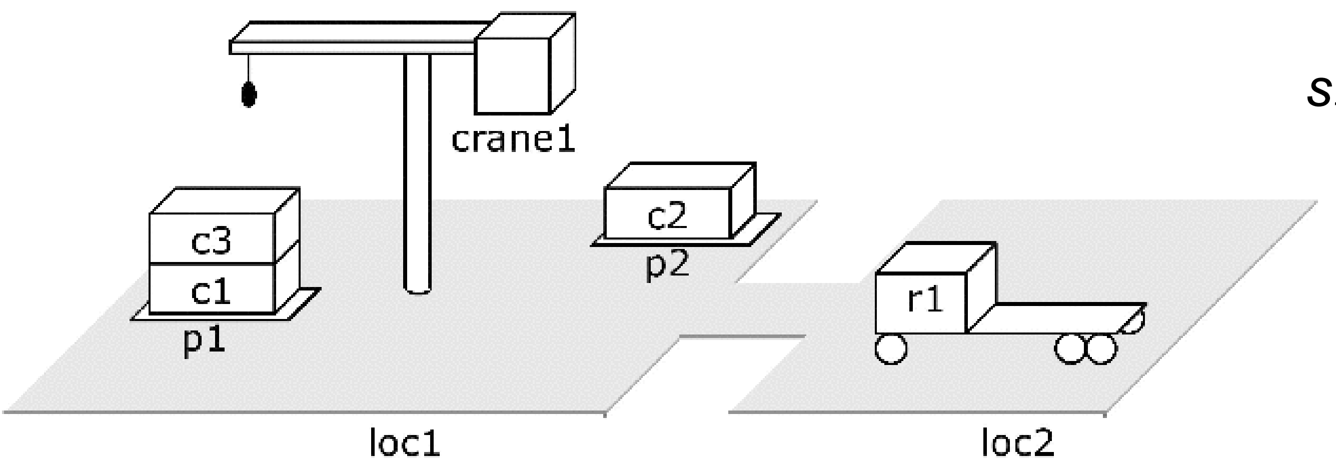


- Solution?

State-Variable Representation

- Use ground atoms for properties that do not change, e.g., adjacent(loc1,loc2)
- For properties that can change, assign values to **state variables**
  - Like fields in a record structure
- Classical and state-variable representations take similar amounts of space
  - Each can be translated into the other in low-order polynomial time

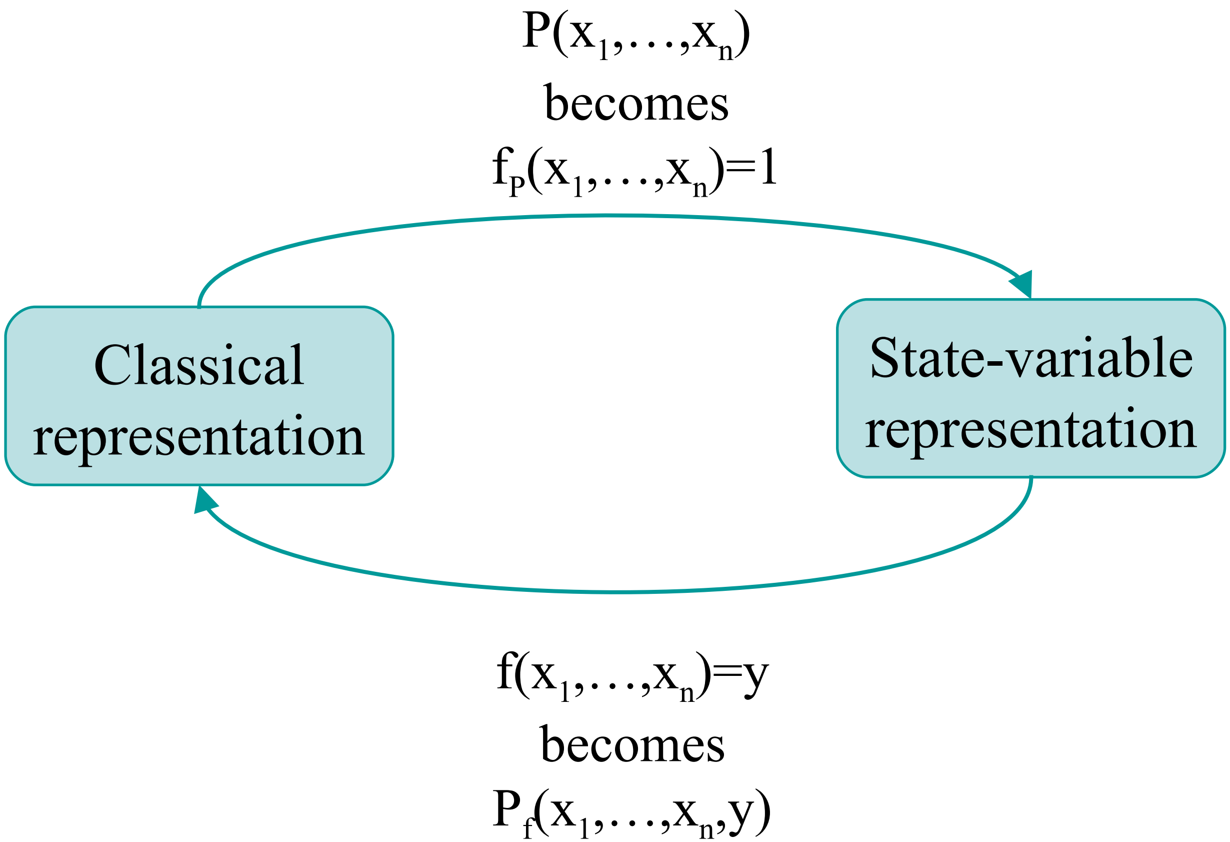
move(r, l, m)
;; robot r at location l moves to an adjacent location m
precond: rloc(r) $= l$, adjacent(l, m)
effects: rloc(r) $\leftarrow m$



s_1 = {top(p1)=c3,
cpos(c3)=c1,
cpos(c1)=pallet,
holding(crane1)=nil,
rloc(r1)=loc2,
loaded(r1)=nil, …}

Expressive Power

- Any problem that can be represented in one representation can also be represented in the other
- Can convert in linear time and space

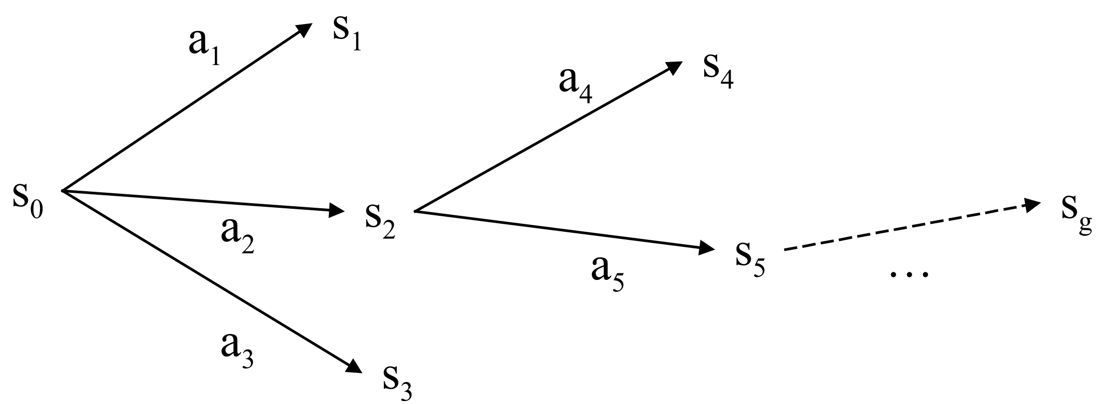


Comparison

- Classical representation
  - The most popular for classical planning, partly for historical reasons
- State-variable representation
  - Equivalent to classical representation in expressive power
  - Less natural for logicians, more natural for engineers and most computer scientists
  - Useful in non-classical planning problems as a way to handle numbers, functions, time

State-Space Planning

Search Algorithms



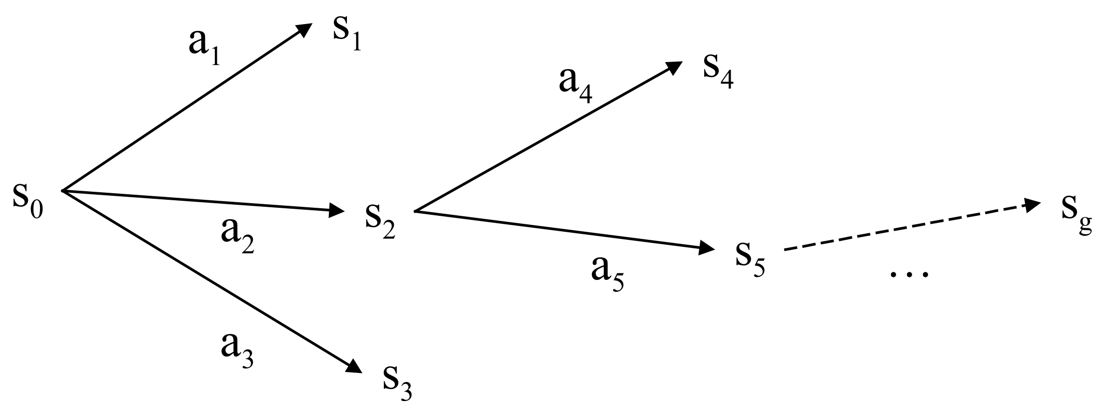
Search tree

- nodes = states
- edges = actions

Search Algorithms

Search tree

- nodes = states
- edges = actions



- **Breadth-first** search is sound and complete
  - But usually not practical because it requires too much memory (exponential in the length of the solution)

- In practice, more likely to use **depth-first** search
  - Worst-case memory requirement is linear in the length of the solution
  - In general, sound but not complete
    - But classical planning has only finitely many states

      ⇨ can make depth-first search complete by doing loop-checking

Exercise

Exercise: Interchange Values of Variables

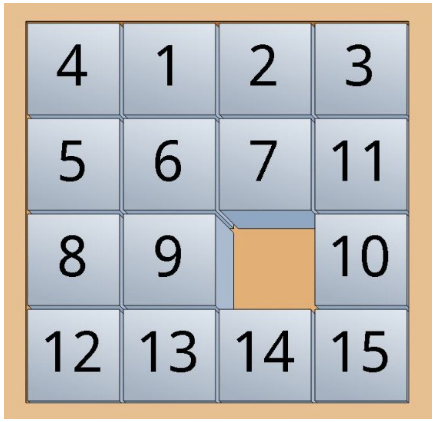
- Operator `assign(v,w,x,y)`
  ```
  precond:  value(v,x), value(w,y)
  effects:  ¬value(v,x), value(v,y)
  ```

- Initial state s_0 = { value(a,3), value(b,5), value(c,0) }
- Goal g = { value(a,5), value(b,3) }

- In the search tree for this planning problem,
  - what is the length of the shortest path to a solution?
  - what is the length of the longest path in the tree?

Planning with Heuristic Search

- Explicitly search with heuristic h(s) that estimates cost from s to goal
- General idea:
  heuristic function = length of optimal plan for a **relaxed problem**
- Examples:
  - Manhattan distance in 15-puzzle
  - Euclidean distance in route finding



- How to get and solve suitable relaxations?
- How to get heuristics automatically?

Heuristics for Classical Planning

- Automatic extraction of informative heuristic function **from the problem P itself**

- Most common relaxation in planning: **ignore all negative effects** of the operators.

  Let P^+ be obtained from planning problem P by dropping the negative effects.

  If $c^*(P^+,s)$ is optimal cost of P^+ with initial state s, then the heuristic is set to

  $$h(s) = c^*(P^+,s)$$

- This heuristic is intractable in general, but easy to approximate

Example.

- Operator `assign(v,w,x,y)`

```
              precond:  value(v,x), value(w,y)
              effects:  ~~¬value(v,x),~~ value(v,y)
```

- s_0 = { value(a,3), value(b,5), value(c,0) }, g = { value(a,5), value(b,3) }
- Optimal relaxed plan: assign(a,b,3,5), assign(b,a,5,3), hence $h(s_0) = 2$

Example

- Operator `assign(v,w,x,y)`

```
precond:  value(v,x), value(w,y)
effects:  ¬value(v,x), value(v,y)
```

- g = { value(a,5), value(b,3) }
- s_0 = { value(a,3), value(b,5), value(c,0) }

Consider all possible successor states after one action:

s_1 = { value(a,5), value(b,5), value(c,0) } $h(s_1) = \infty$

No relaxed plan exists

s_2 = { value(a,3), value(b,3), value(c,0) } $h(s_2) = \infty$

s_3 = { value(a,0), value(b,5), value(c,0) } $h(s_3) = \infty$

s_4 = { value(a,3), value(b,5), value(c,3) } $h(s_4) = 2$

s_5 = { value(a,3), value(b,0), value(c,0) } $h(s_5) = \infty$

s_6 = { value(a,3), value(b,5), value(c,5) } $h(s_6) = 2$

Example

- Operator `assign(v,w,x,y)`
  ```
  precond:  value(v,x), value(w,y)
  effects:  ¬value(v,x), value(v,y)
  ```
- g = { value(a,5), value(b,3) }
- s_4 = { value(a,3), value(b,5), value(c,3) }

Consider all possible successor states after next action:

s_7 = { value(a,5), value(b,5), value(c,3) } $h(s_1) = 1$

s_8 = { value(a,3), value(b,3), value(c,3) } $h(s_8) = \infty$

s_9 = { value(a,3), value(b,5), value(c,5) } $h(s_9) = 2$

One of the successor states of s_7 is a goal state:

s_{10} = { value(a,5), value(b,3), value(c,3) }

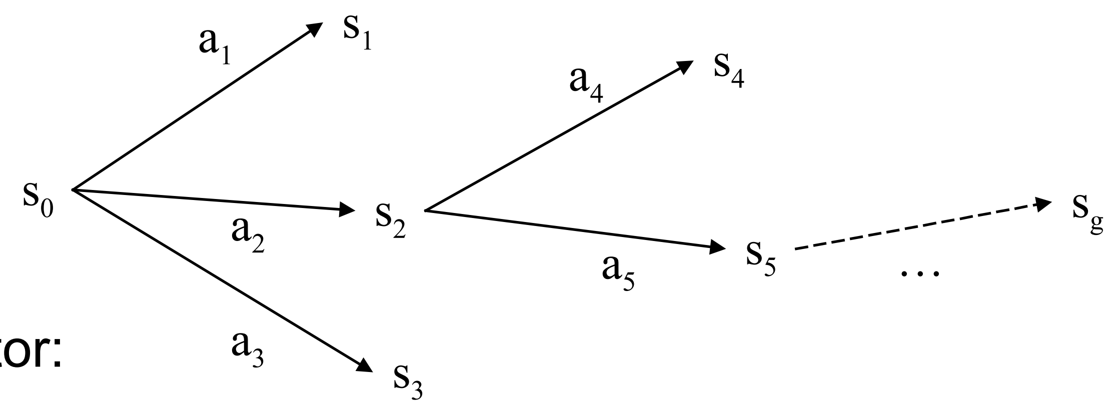
Planning-Graph Techniques

History

- Before Graphplan came out, most planning researchers were working on Plan Space Search-like planners
- **Graphplan** caused a sensation because it was so much faster
- Many subsequent planning systems have used ideas from it
  - IPP, STAN, GraphHTN, SGP, Blackbox, Medic, TGP, LPG
  - Many of them even much faster than the original Graphplan

Motivation

- A big source of inefficiency in search algorithms is the **branching factor** (= the number of children of each node)
- A standard tree search may try lots of actions that are unrelated to the goal



- One way to reduce branching factor:
- 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, …
  - *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

Example: Have the Cake and Eat it Too

Operator Name	Preconditions	Effects
eat(c)	have(c)	¬have(c), eaten(c)
bake(c)	¬have(c)	have(c)
- Also have the maintenance actions: one for each literal
- s0 = { have(cake) }
- g = { have(cake), eaten(cake) }

state-level 0

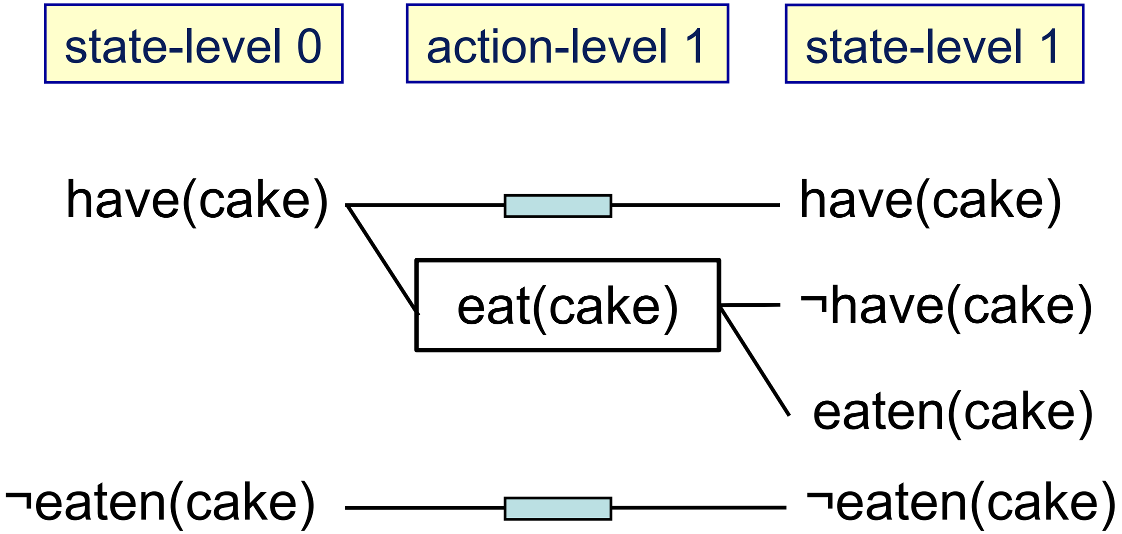
have(cake)

¬eaten(cake)

Example: Have the Cake and Eat it Too

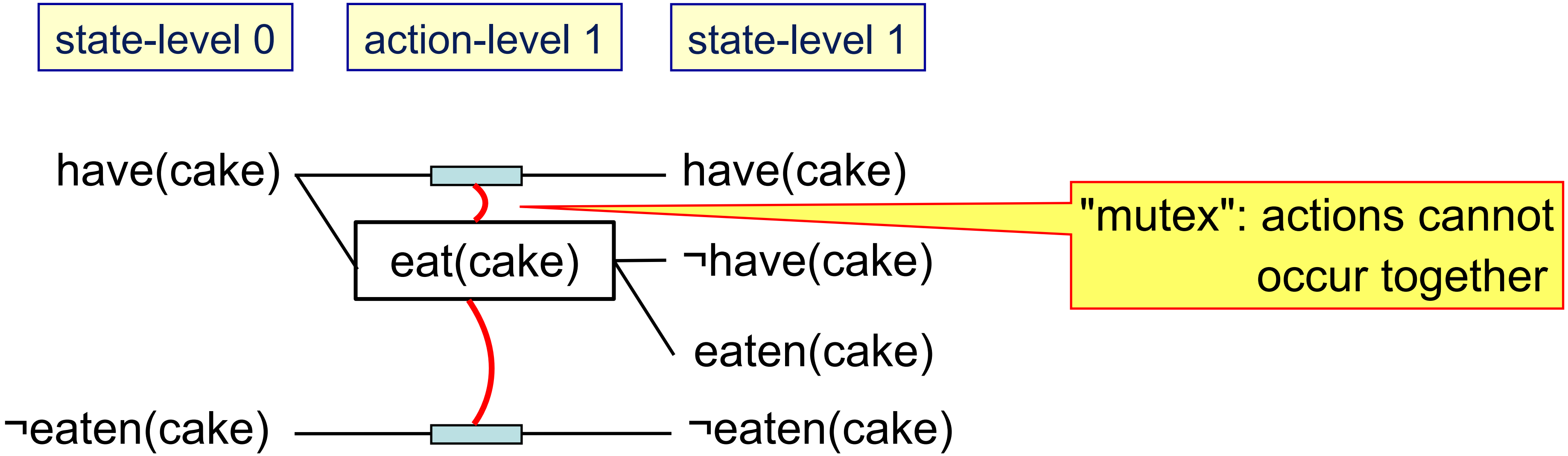
Operator Name	Preconditions	Effects
eat(c)	have(c)	¬have(c), eaten(c)
bake(c)	¬have(c)	have(c)

- Also have the maintenance actions: one for each literal
- s0 = { have(cake) }
- g = { have(cake), eaten(cake) }



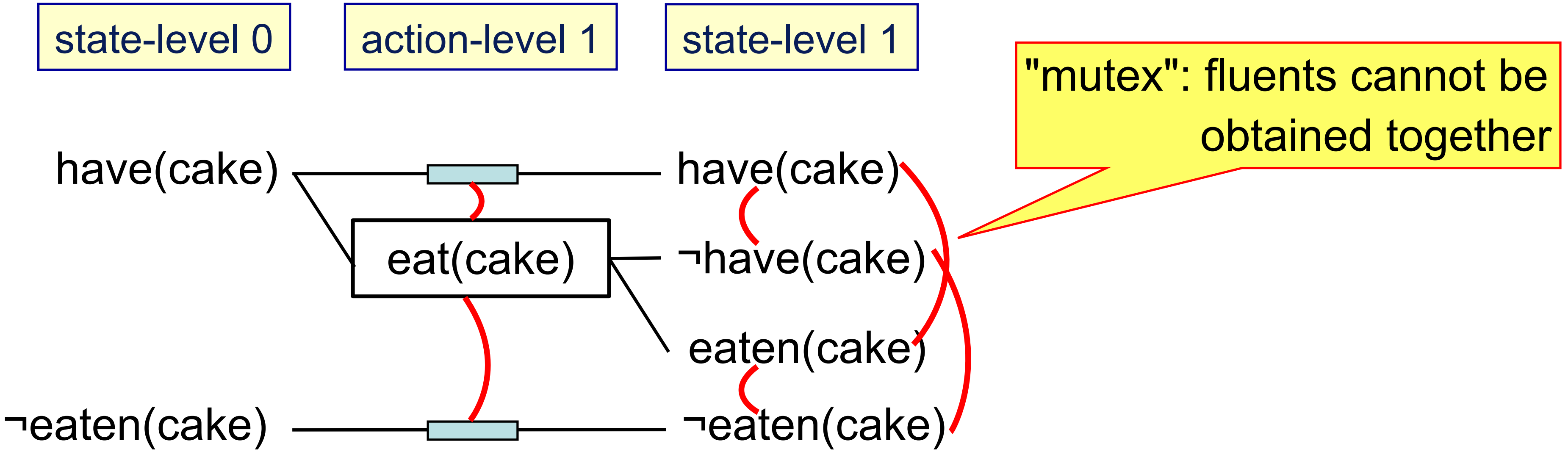
Example: Have the Cake and Eat it Too

Operator Name	Preconditions	Effects
eat(c)	have(c)	¬have(c), eaten(c)
bake(c)	¬have(c)	have(c)
- Also have the maintenance actions: one for each literal
- s0 = { have(cake) }
- g = { have(cake), eaten(cake) }



Example: Have the Cake and Eat it Too

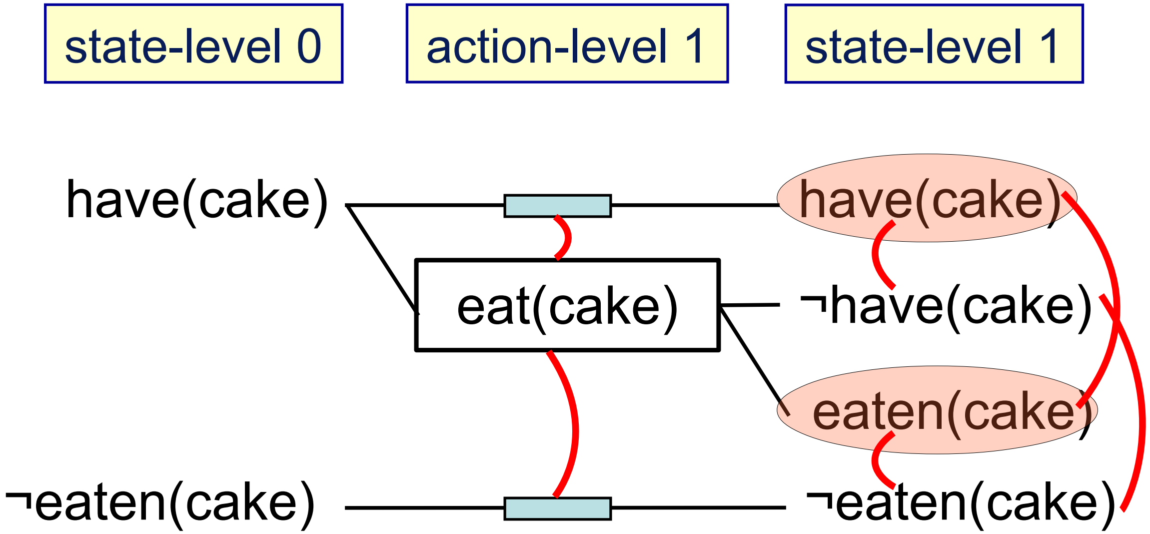
Operator Name	Preconditions	Effects
eat(c)	have(c)	¬have(c), eaten(c)
bake(c)	¬have(c)	have(c)
- Also have the maintenance actions: one for each literal
- s0 = { have(cake) }
- g = { have(cake), eaten(cake) }



Example: Have the Cake and Eat it Too

Operator Name	Preconditions	Effects
eat(c)	have(c)	¬have(c), eaten(c)
bake(c)	¬have(c)	have(c)

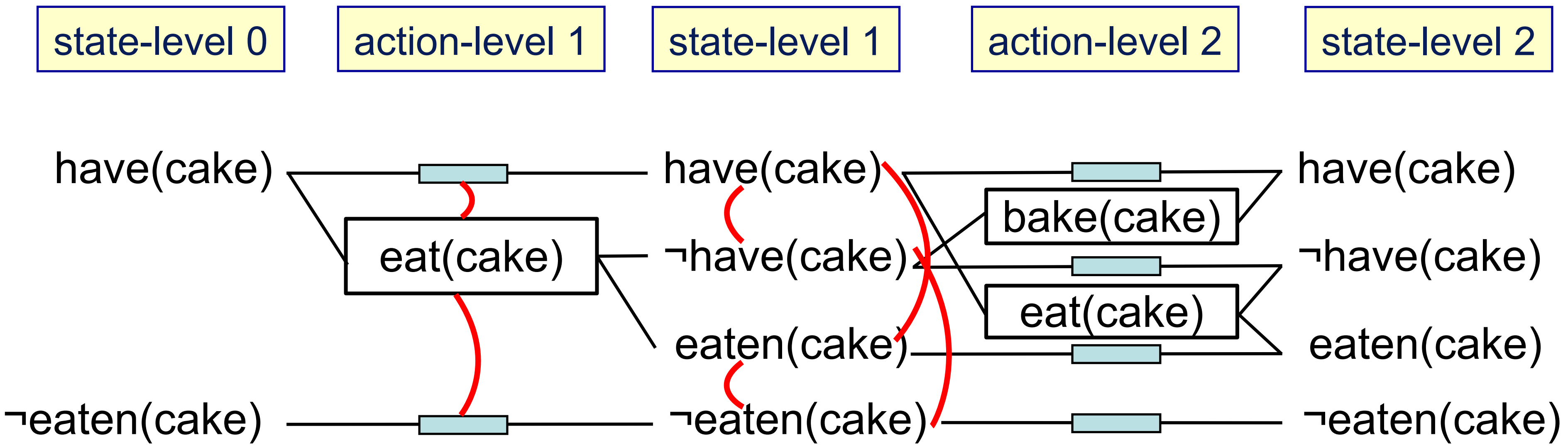
- Also have the maintenance actions: one for each literal
- s0 = { have(cake) }
- g = { have(cake), eaten(cake) }



Solution extraction **not** called since goals are mutex

Example: Have the Cake and Eat it Too

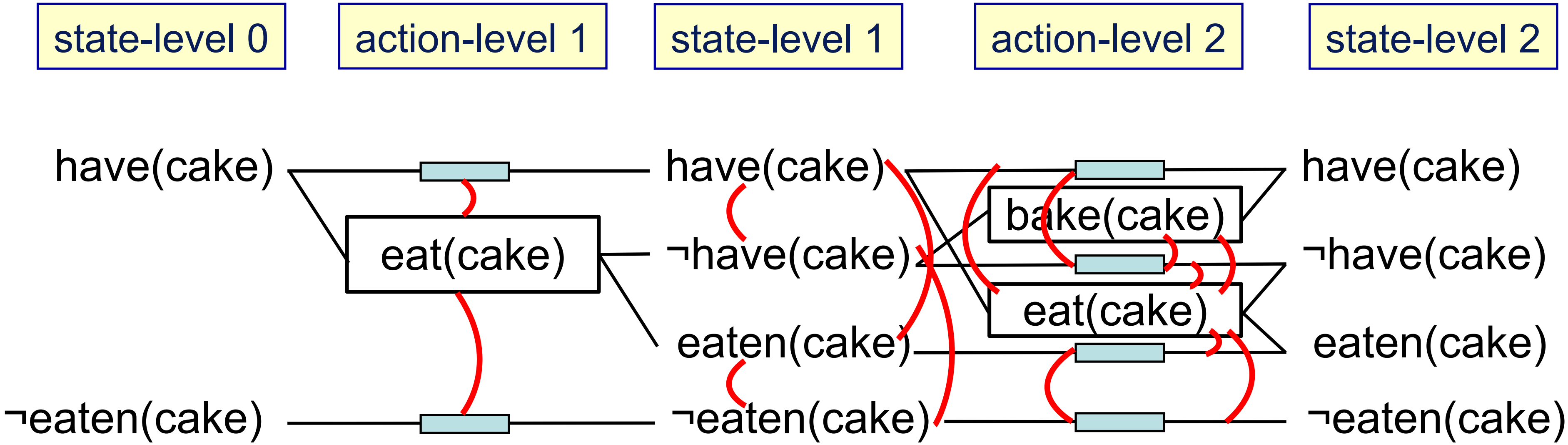
Operator Name	Preconditions	Effects
eat(c)	have(c)	¬have(c), eaten(c)
bake(c)	¬have(c)	have(c)
- Also have the maintenance actions: one for each literal
- s0 = { have(cake) }
- g = { have(cake), eaten(cake) }



Example: Have the Cake and Eat it Too

Operator Name	Preconditions	Effects
eat(c)	have(c)	¬have(c), eaten(c)
bake(c)	¬have(c)	have(c)

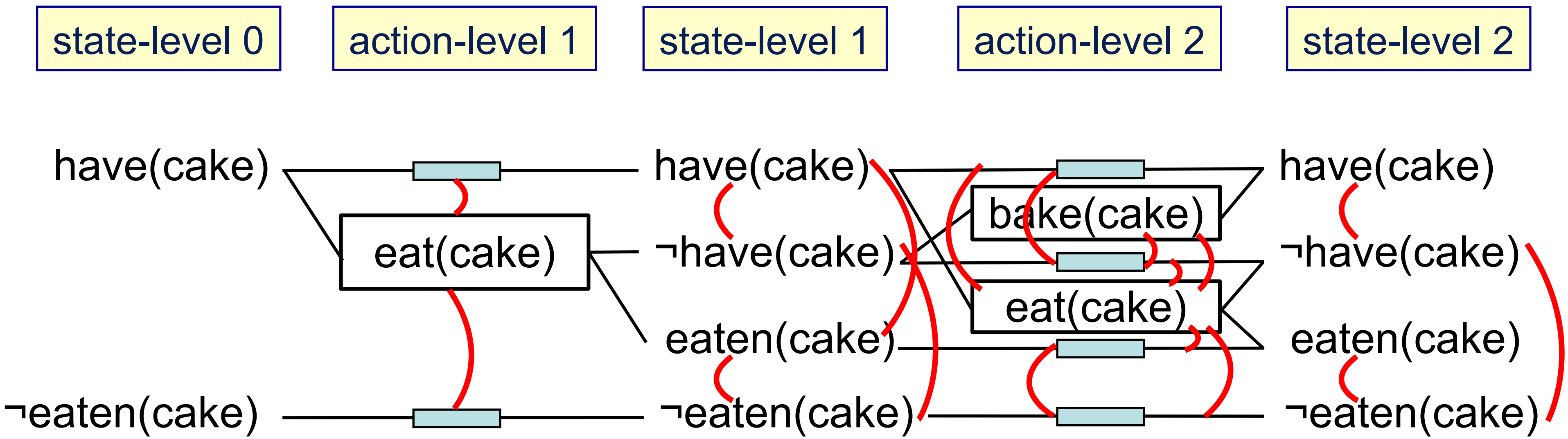
- Also have the maintenance actions: one for each literal
- s0 = { have(cake) }
- g = { have(cake), eaten(cake) }



Example: Have the Cake and Eat it Too

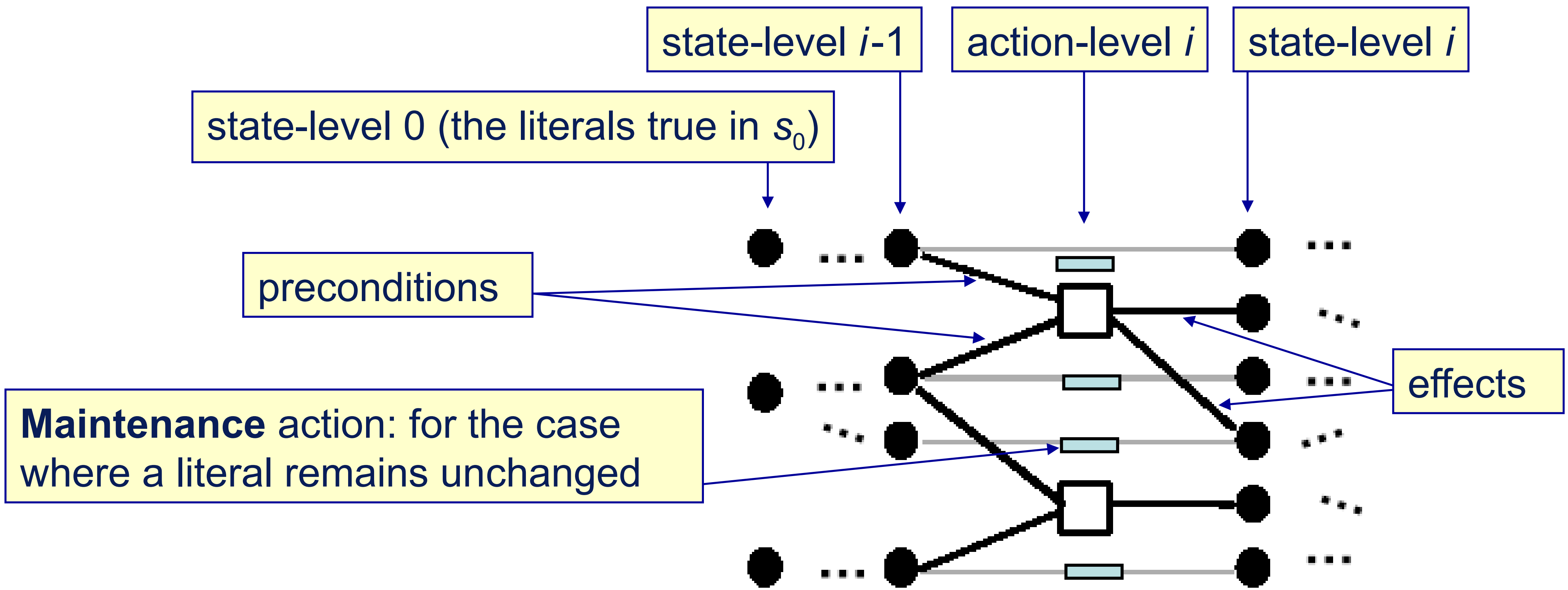
Operator Name	Preconditions	Effects
eat(c)	have(c)	¬have(c), eaten(c)
bake(c)	¬have(c)	have(c)

- Also have the maintenance actions: one for each literal
- s0 = { have(cake) }
- g = { have(cake), eaten(cake) }

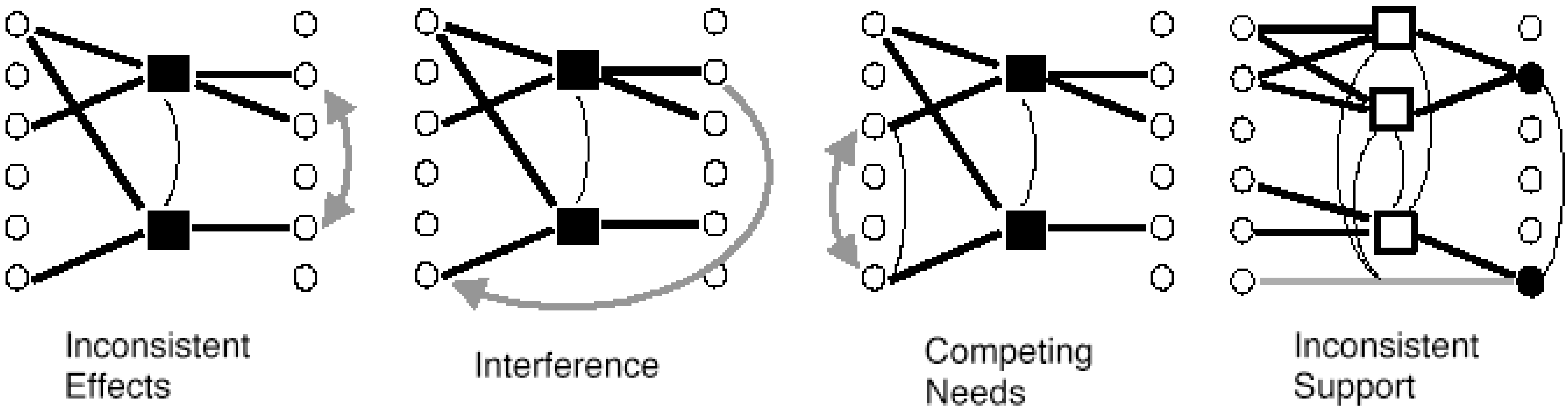


The Planning Graph

- 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



Mutual Exclusion



- Two actions at the same action-level are mutex if
  1. **Inconsistent effects:** an effect of one negates an effect of the other
  2. **Interference:** one deletes a precondition of the other
  3. **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
  4. **Inconsistent support:** one is the negation of the other, **or all ways of achieving them are pairwise mutex**

Recursive propagation of mutexes

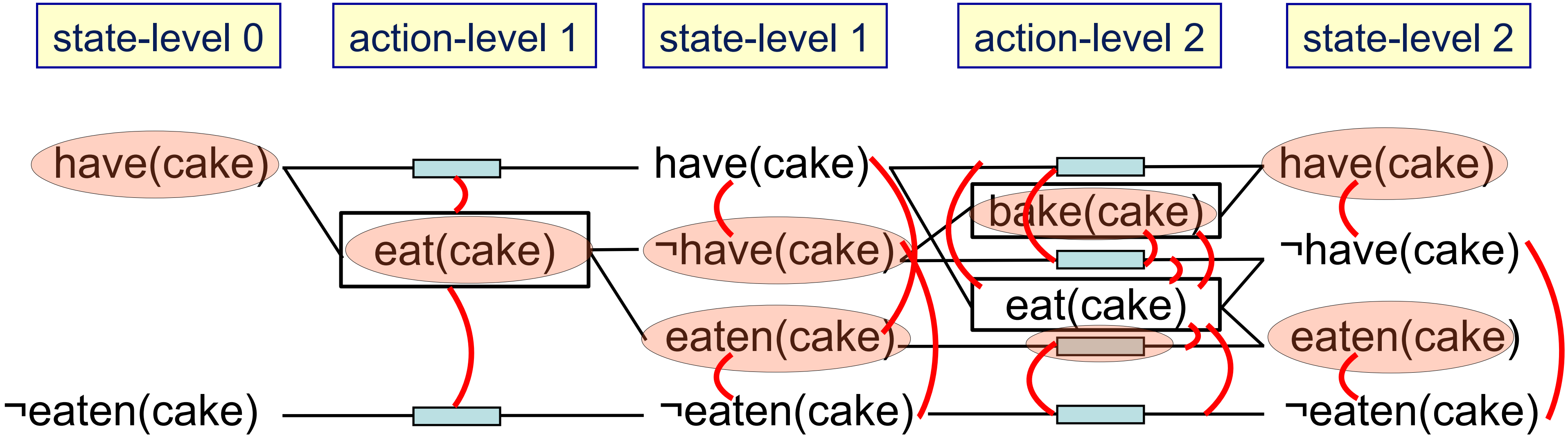
Mutexes in the Cake-Example

Level	Mutexes		Rule
A1	*eat(cake)*	$m_{have(cake)}$	1 (also 2)
A1	*eat(cake)*	$m_{\neg eaten(cake)}$	1, 2
S1	*have(cake)*	*¬have(cake)*	4
S1	*eaten(cake)*	*¬eaten(cake)*	4
S1	*have(cake)*	*eaten(cake)*	4
S1	*¬have(cake)*	*¬eaten(cake)*	4
A2	*bake(cake)*	*eat(cake)*	1, 3
A2	*bake(cake)*	$m_{\neg have(cake)}$	1, 2
A2	*eat(cake)*	$m_{have(cake)}$	1, 2
A2	*eat(cake)*	$m_{\neg have(cake)}$	3
A2	*eat(cake)*	$m_{eaten(cake)}$	**3**
A2	*eat(cake)*	$m_{\neg eaten(cake)}$	1, 2
A2	$m_{have(cake)}$	$m_{\neg have(cake)}$	1, 2, 3
A2	$m_{eaten(cake)}$	$m_{\neg eaten(cake)}$	1, 2, 3
S2	*have(cake)*	*¬have(cake)*	4
S2	*eaten(cake)*	*¬eaten(cake)*	4
S2	*¬have(cake)*	*¬eaten(cake)*	4

Example: Have the Cake and Eat it Too

Solution extraction **succeeds**

(= plan without mutexes)



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*

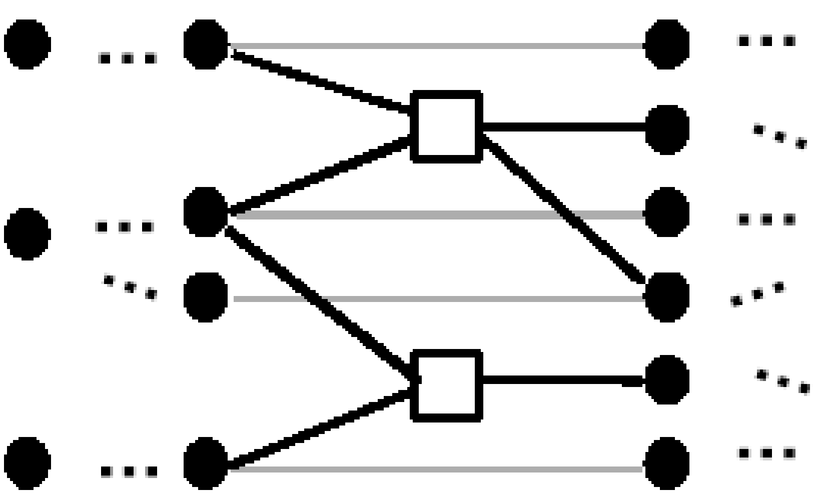
if any pair of chosen actions are mutex

then backtrack

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

Solution-extraction(*g'*, *j*–1)

end Solution-extraction



Comparison with State-Space Planning

- Advantage:
  - The backward-search part (solution extraction) of Graphplan—which is the hard part—will only look at the actions in the planning graph
  - smaller search space than state-space planning; thus faster
- Disadvantage:
  - To generate the planning graph, Graphplan creates a huge number of ground atoms
  - Many of them may be irrelevant
- For classical planning, the advantage outweighs the disadvantage
  - GraphPlan solves classical planning problems much faster than SSP without heuristcs

Summary

- Representations for classical planning
  - Classical representation
  - State-variable representation
- State-space planning
  - with heuristics
- Planning graphs
  - Creating the graph
  - Adding mutexes
  - Searching the graph