CS 221: Artificial Intelligence

Planning
(and Basic Logic)

Peter Norvig and Sebastian Thrun

Slide credits: Stuart Russell, Rina Dechter, Rao Kambhampati

AI: Dealing with Complexity

- Agent Design
  - Reflex→Memory-Based
  - Reflex→Goal-Based→Utility-Based
- Environment
  - →Partially Observable, Stochastic, Dynamic, Multi-agent, Adversarial
- Representation
  - Atomic→Factored→Structured

Finding Actions

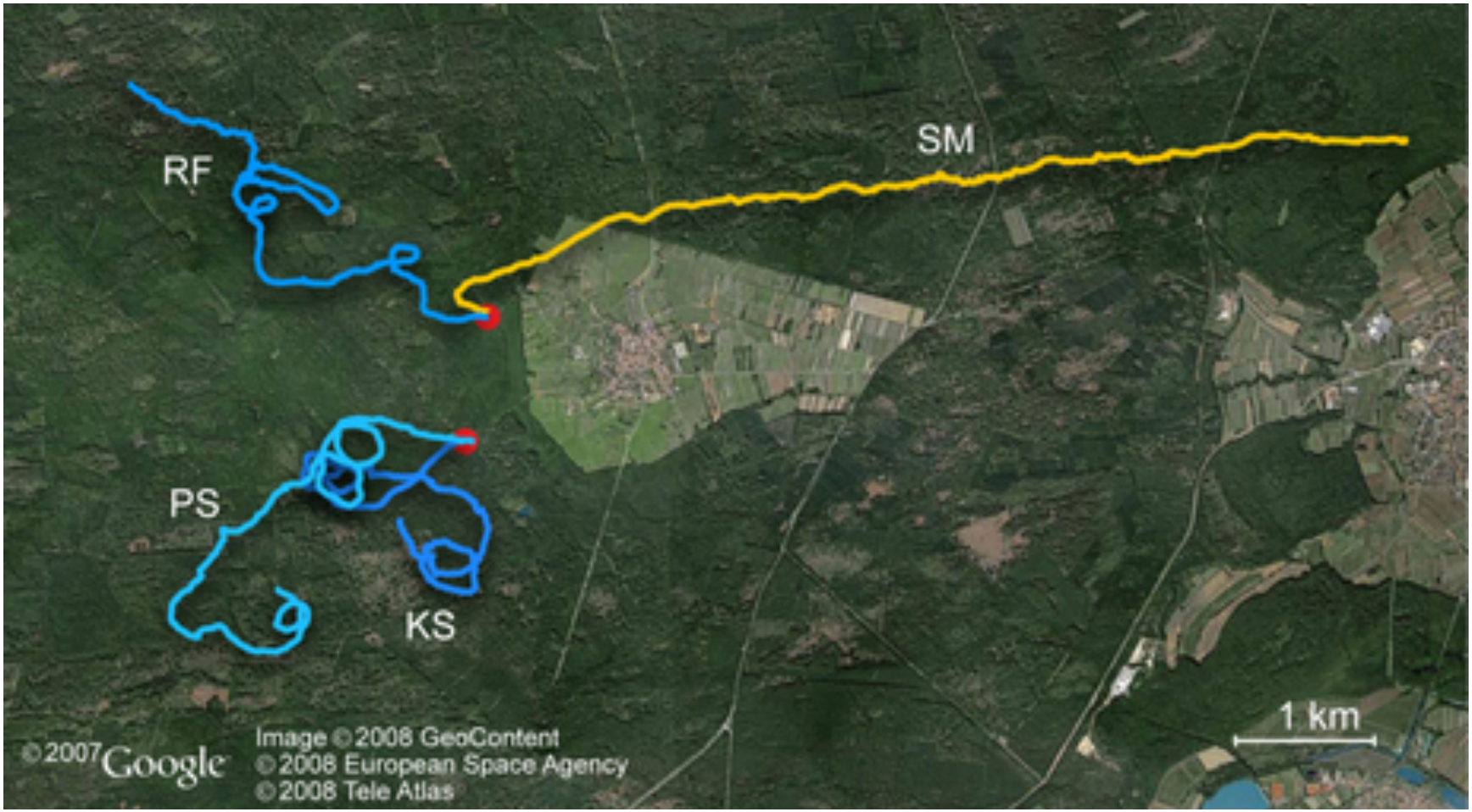
- We've done: **Problem-Solving**

  [Arad, Sibiu, Fagras, Bucharest]

- Introduce today: **Planning**

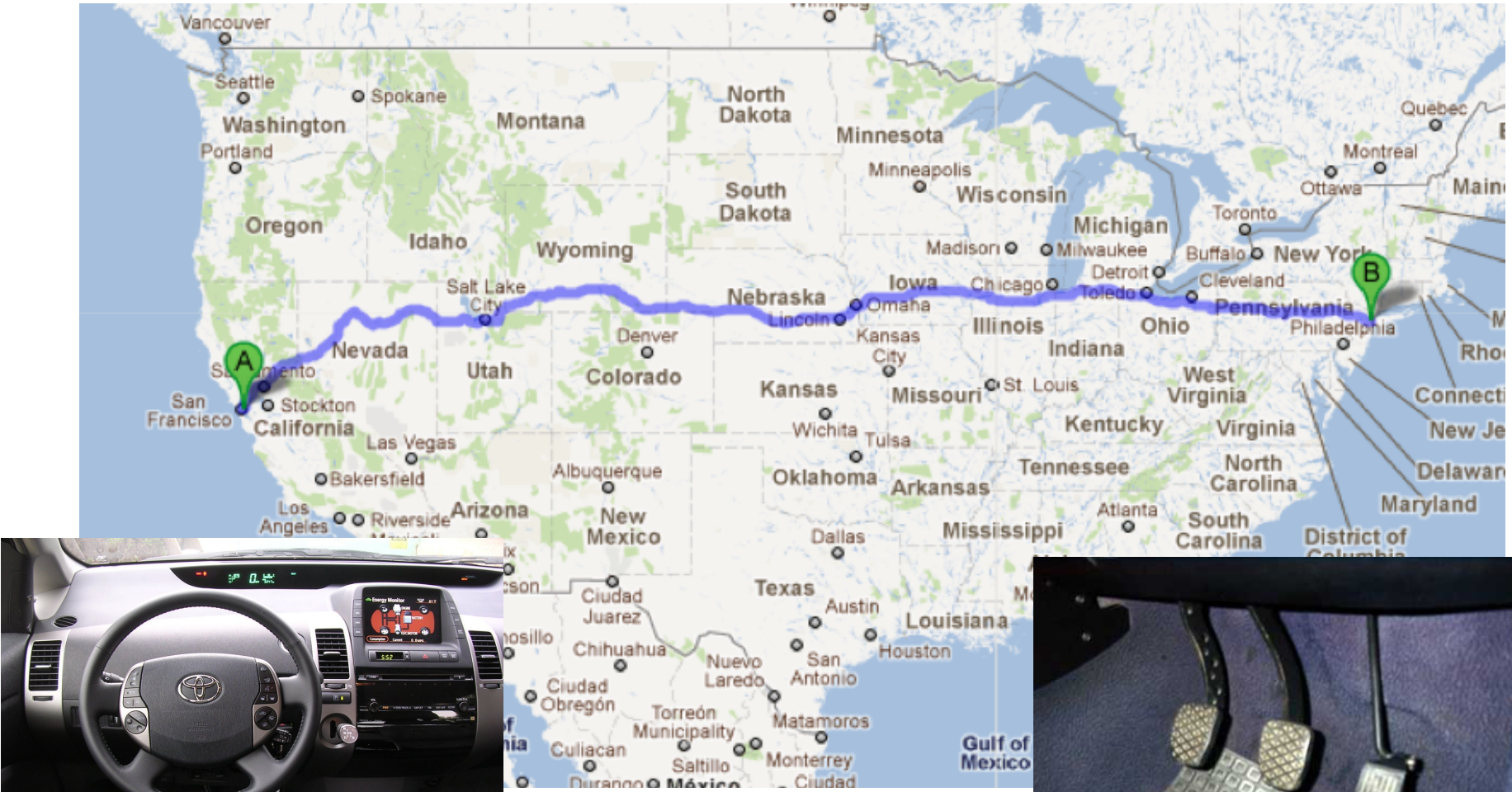
  [while DoorLocked: try another key;
  push door; walk ahead]

What's wrong with Problem-Solving



Plan: [Forward, Forward, Forward, …]

What's wrong with Problem-Solving



Planning

- A plan is a **program**
  - Not just a straight-line sequence of actions
- Planning and acting can be **interleaved**
  - Closed loop, not closed eyes
- Representation is more flexible
  - Can deal with partially-described states
  - Can deal with features, objects, relations
  - Can be hierarchical

Dealing with Partial Observability: World vs. Belief States

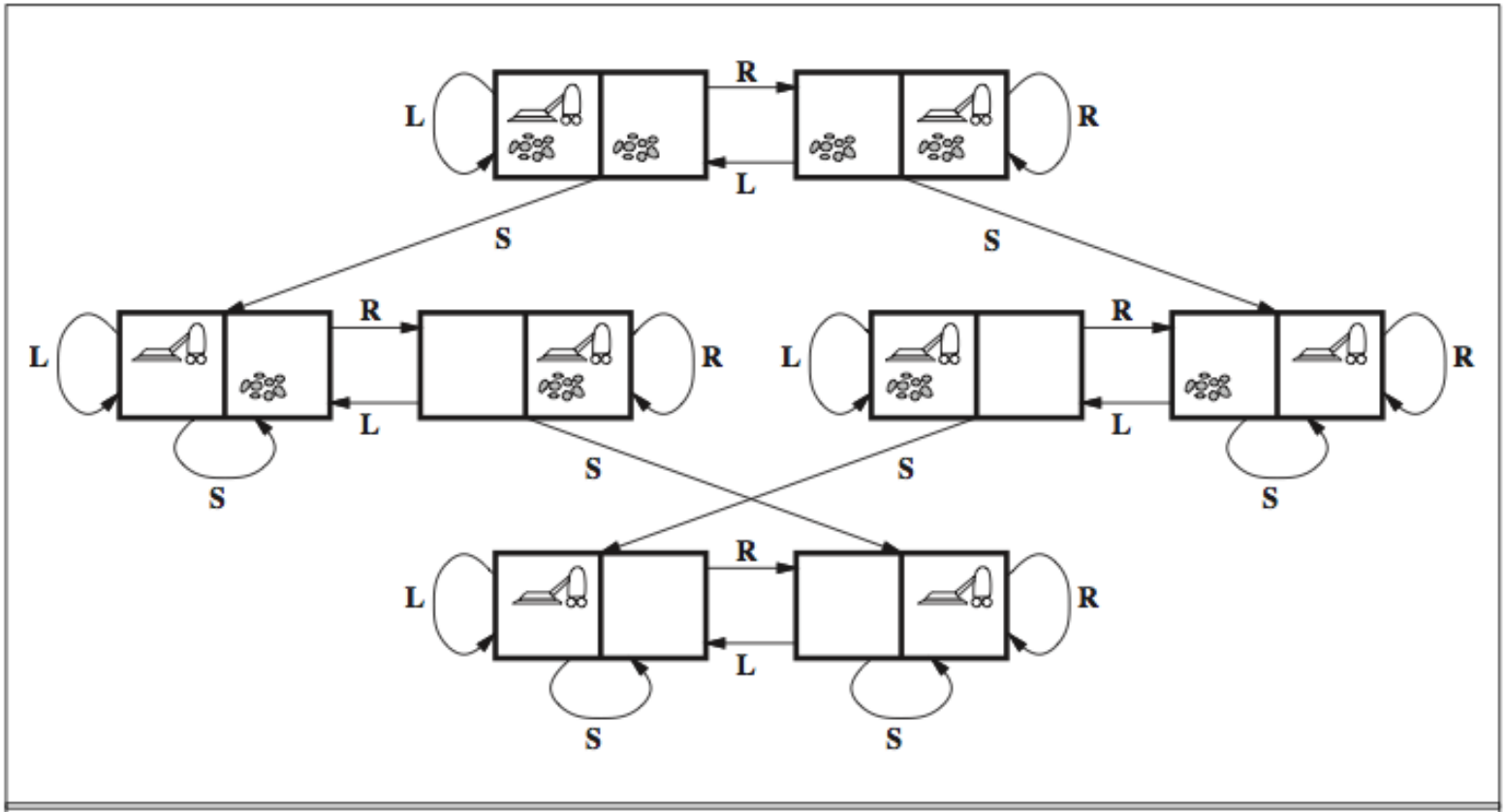
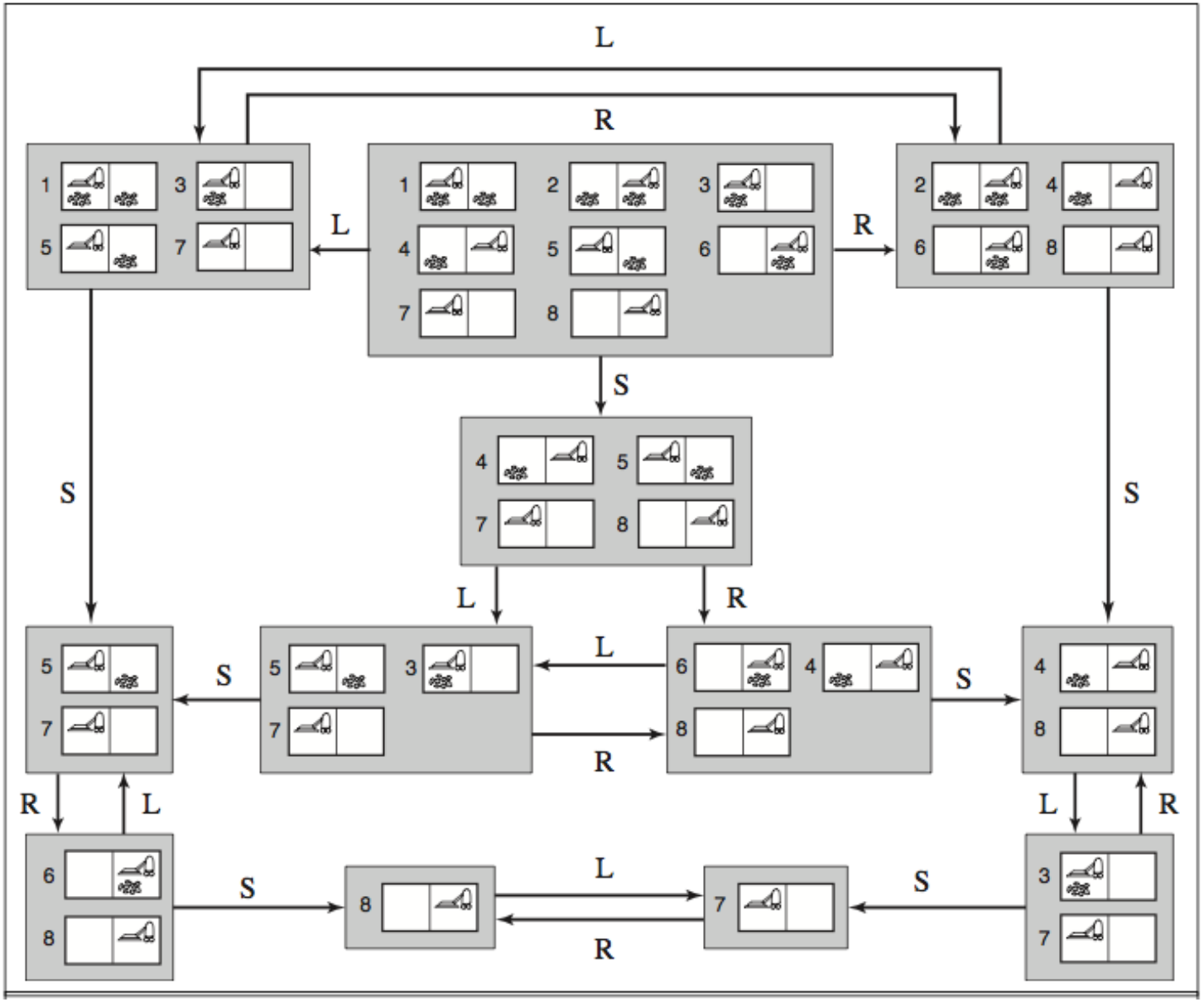
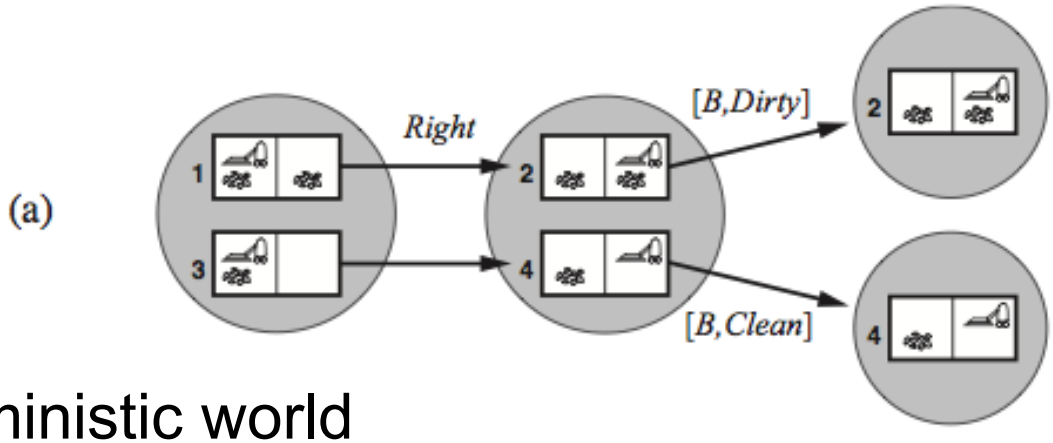


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

Sensorless – Belief States - Conformant Plans

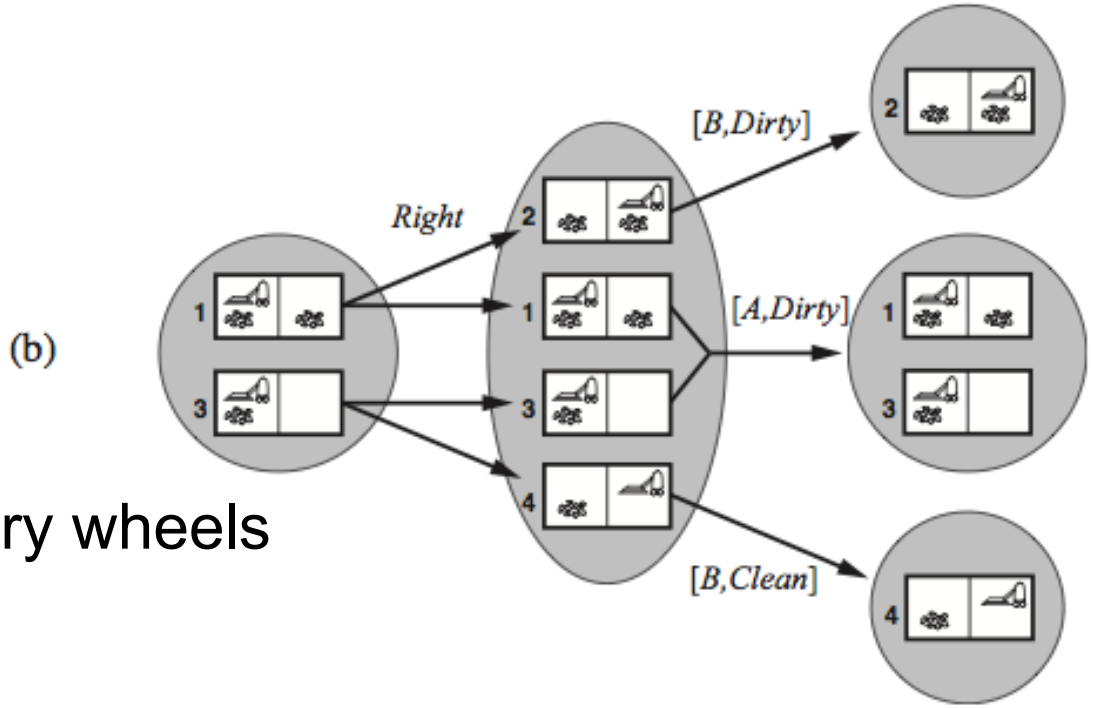


Partial (local) Observability and Stochastic Worlds



Deterministic world

Deterministic actions; observe only local square

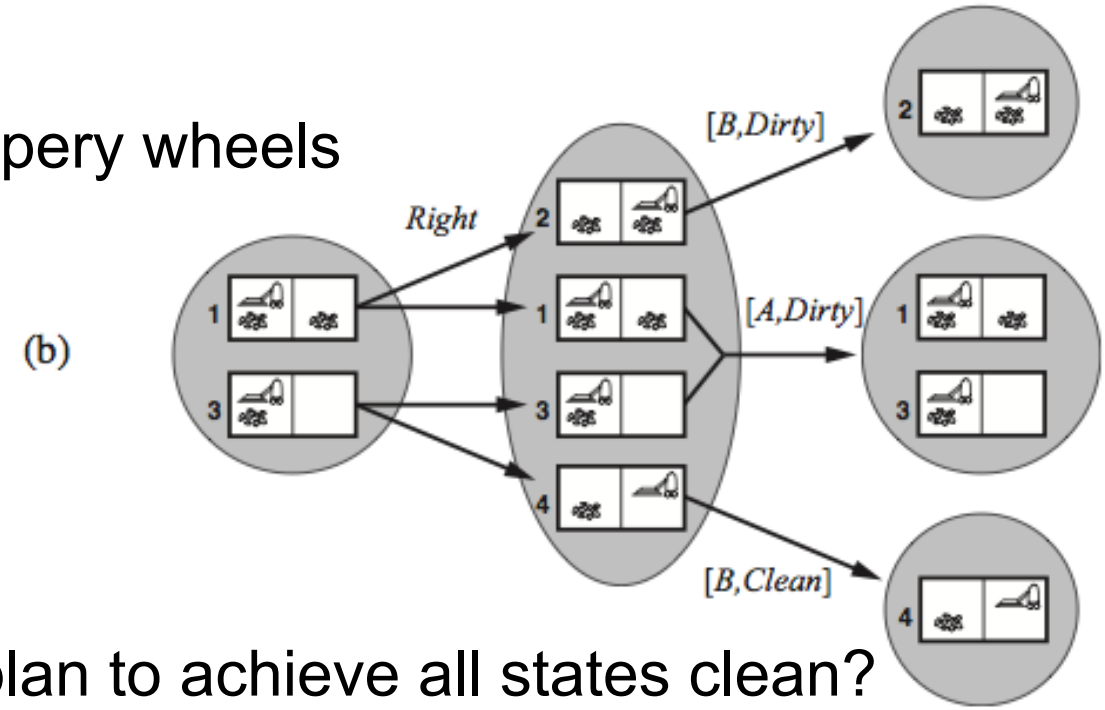


Slippery wheels

Observe only local square; *Suck* is determ. *R/L* are stoch. (may fail to move)

Planning and Sensing in Partially Observable and Stochastic World

Slippery wheels



Observe only local square; *Suck* is determ. *R/L* are stoch.; may fail to move

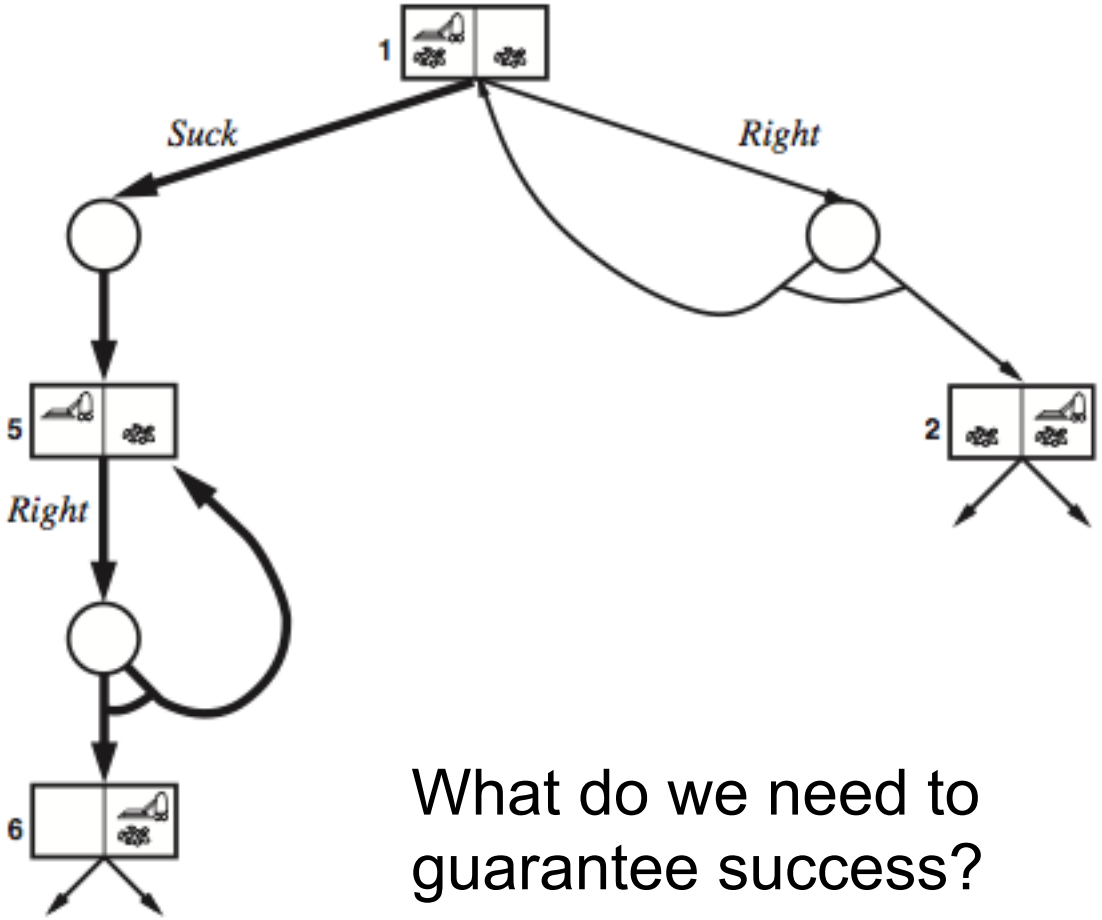
What is a plan to achieve all states clean?

[1:*Suck*; 2:*Right*; (**if** *A*: **goto** 2); 3:*Suck*]

also written as

[*Suck*; (**while** *A*: *Right*); *Suck*]

Search Graph as And/Or Tree



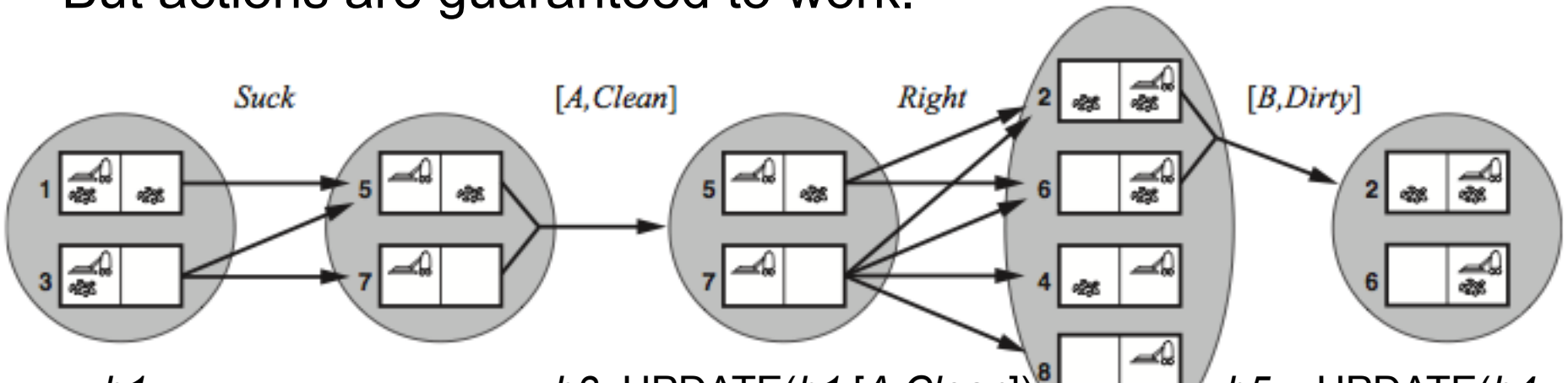
What do we need to guarantee success?
What kind of guarantee?

As Equations, not Tree

- *b* is a belief state: a set of states
- *o* is an observation; a percept
- *a* is an action

- *b'* = UPDATE(PREDICT(*b*, *a*) , *o*)

Kindergarten world: dirt may appear anywhere at any time,
But actions are guaranteed to work.



b1

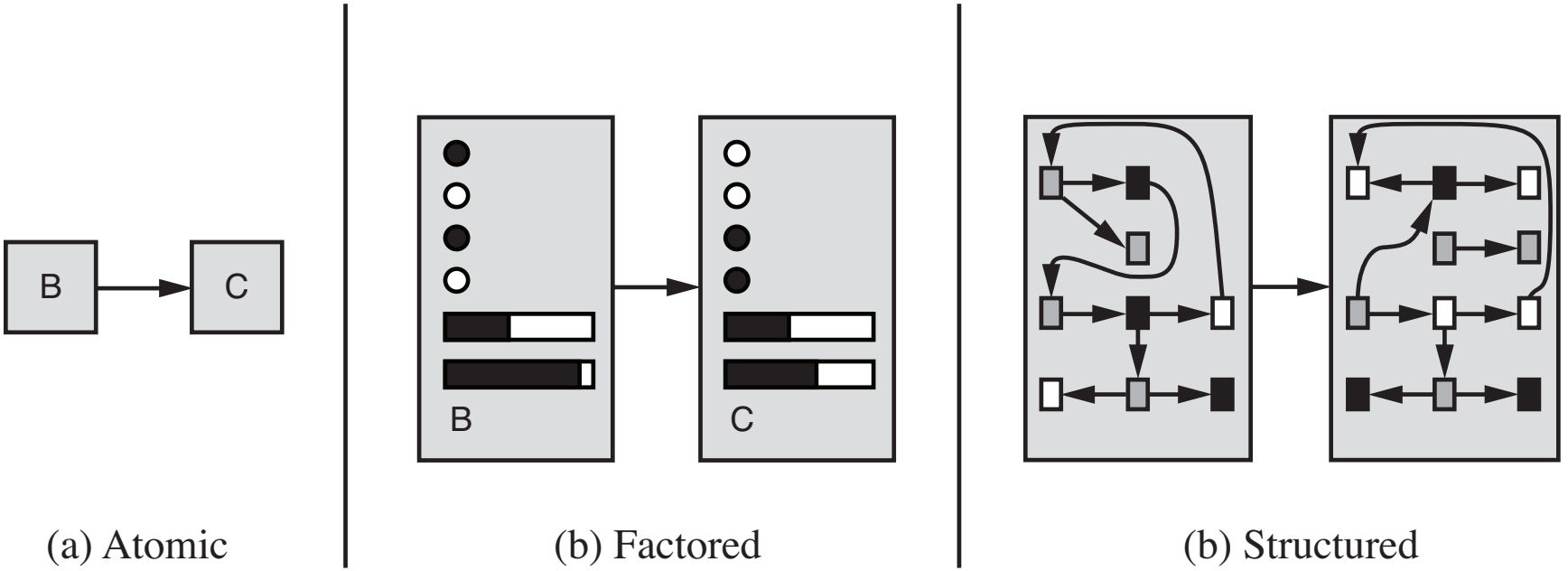
b3=UPDATE(*b1*,[*A,Clean*])

b5 = UPDATE(*b4,…*)

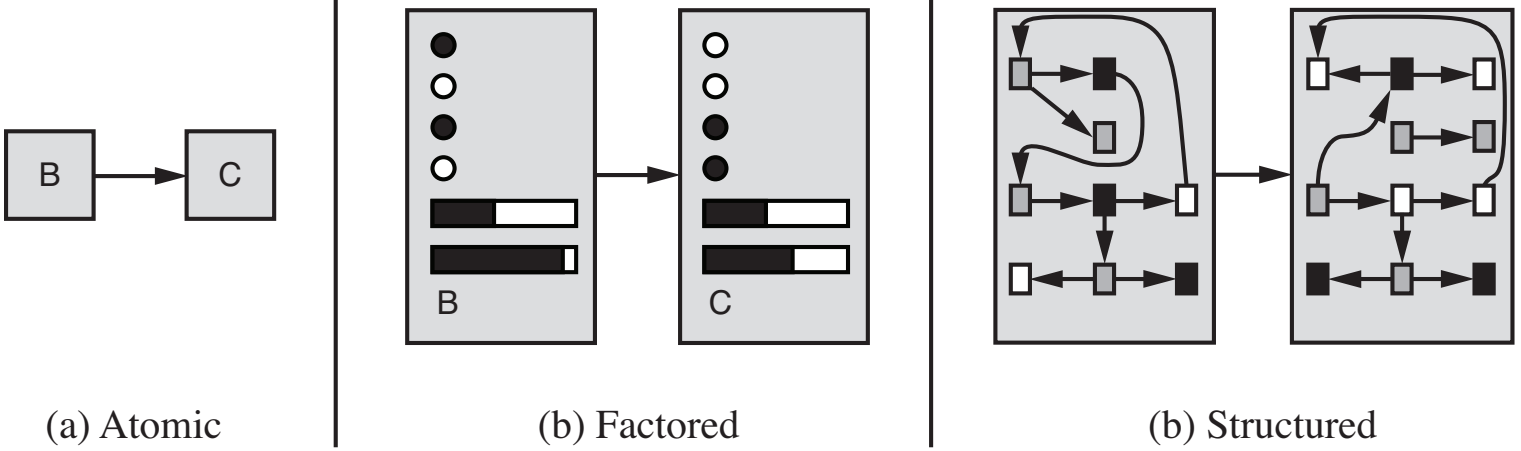
b2=PREDICT(*b1, Suck*)

b4=PREDICT(*b3*,*Right*)

State Representation



State Representation



- Bayes Nets?

  Factored
- SQL data base?

  One table: Factored; Several: Structured
- Java program?

  Structured

Representation

- Atomic Representation
  - $s_1 = s_2$; Result(*s, a*) = *s′*; GoalTest(*s*);
- Factored Representation
  - Attribute(*s*) = *val …* (*val* numeric or Boolean)
  - Result and GoalTest in terms of attributes
- Structured Representation
  - All of above
  - Relations, Functions: Rel(*a*, *b*, *c*); F(*a*, *b*) = *c*
  - Objects; with Quantifiers
    (for all ∀x, there exists ∃y)

Planning with Factored States

- World is made up of states which are defined in terms of state variables
  - Can be Boolean or categorical or continuous
  - Where have we used this before?
- State: *complete assignment over state variables*
  So, *k* Boolean state variables represent how many states?
- Actions change the values of the state variables
  - Applicability conditions of actions are also specified in terms of partial assignments over state variables

"Classical" Planning

- **State**: conjunction of Boolean state variables
- **Action Schema**:
  - *Action*(*Fly*(*p, from, to*),
    Precond: *At*(*p*, *from*) ∧ *Plane*(*p*)
    ∧ *Airport*(*from*) ∧ *Airport*(*to*)
    Effect: ¬*At*(*p*, *from*) ∧ *At*(*p*, *to*))

Implicitly defines *Actions*(*s*) and *Result*(*s, a*)

Expressiveness of the language

$Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK)$
$\quad \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2)$
$\quad \land Airport(JFK) \land Airport(SFO))$
$Goal(At(C_1, JFK) \land At(C_2, SFO))$
$Action(Load(c, p, a),$
$\quad$ PRECOND: $At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
$\quad$ EFFECT: $\neg At(c, a) \land In(c, p))$
$Action(Unload(c, p, a),$
$\quad$ PRECOND: $In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
$\quad$ EFFECT: $At(c, a) \land \neg In(c, p))$
$Action(Fly(p, from, to),$
$\quad$ PRECOND: $At(p, from) \land Plane(p) \land Airport(from) \land Airport(to)$
$\quad$ EFFECT: $\neg At(p, from) \land At(p, to))$

Figure 10.1 A PDDL description of an air cargo transportation planning problem.

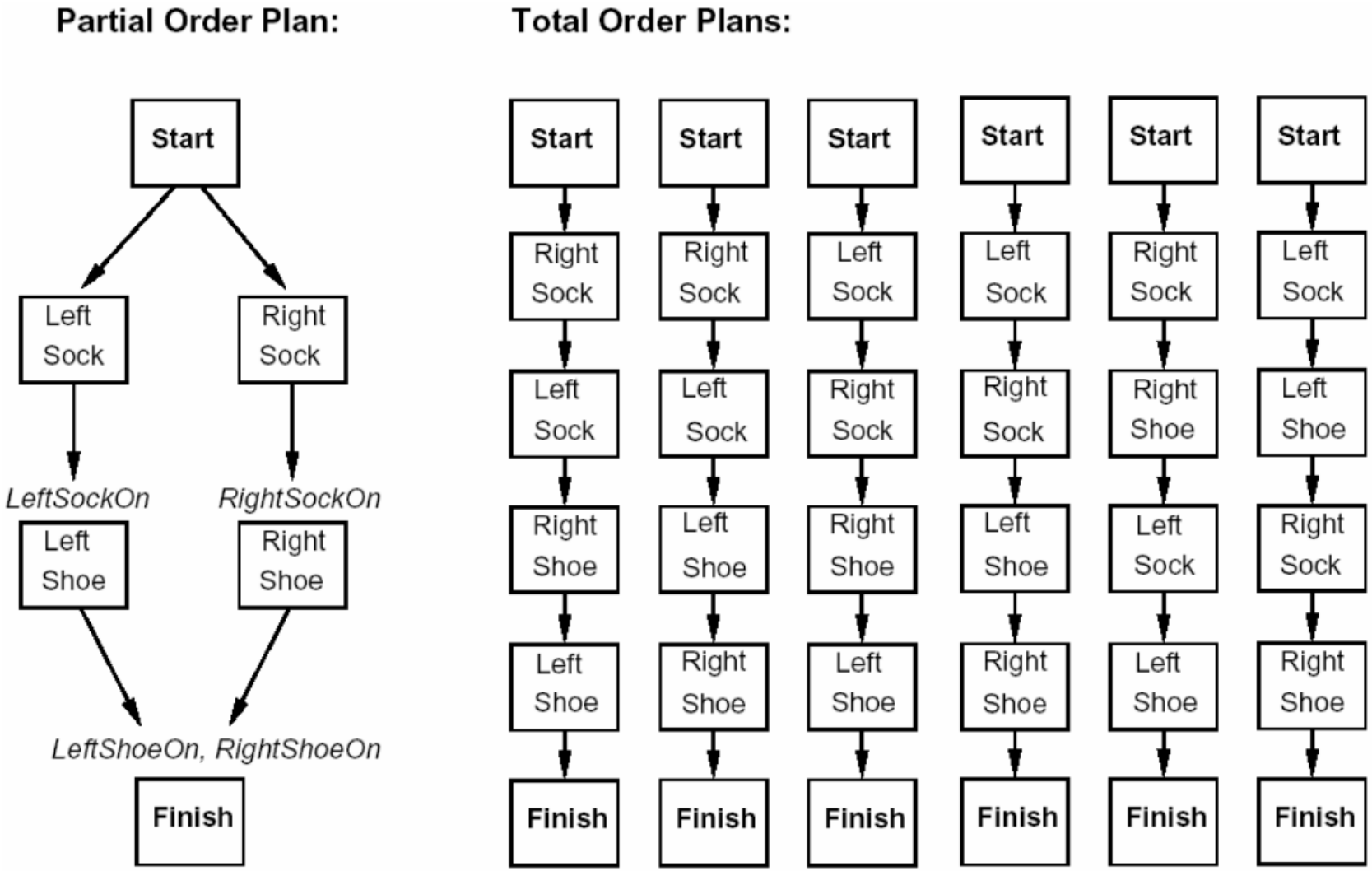
Advantages of the Language

- Natural to read, write, verify
- Abstract over similar actions
- Easy to extend with more complex syntax and semantics
- Can be compiled or interpreted
- Can automatically derived heuristics (relaxed problem)

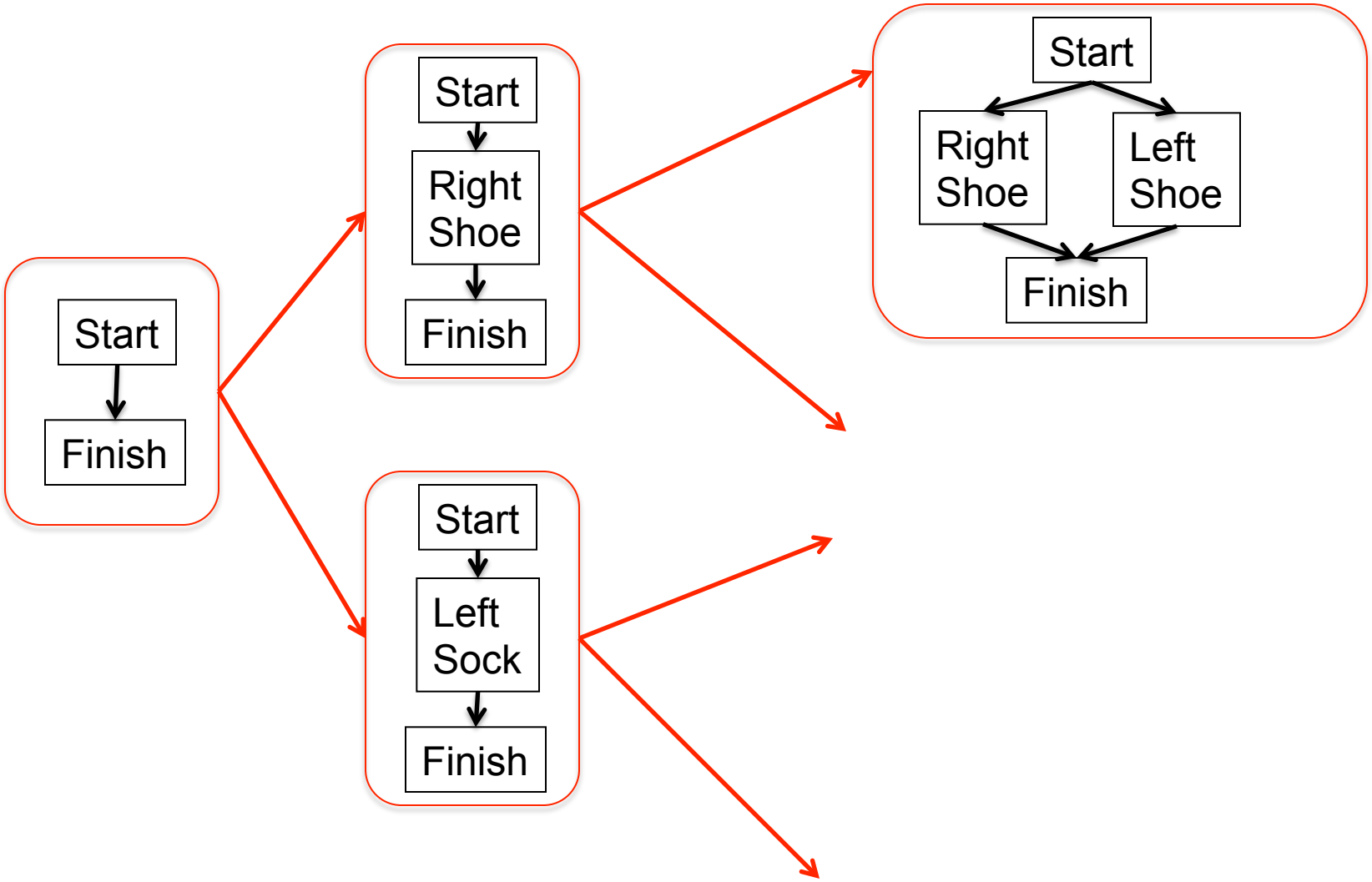
Planning Algorithms

- Forward (progression) state-space search
  - … it's just search
- Backward (regresssion) state-space search
  - Consider Goal: *Own*(0136042597)
    Action(*Buy*(*i*), Pre: *ISBN*(*i*) Eff: *Own*(*i*))
  - In general, may involve unbound variables
- Plan-space search
  - Start with empty plan, add branches

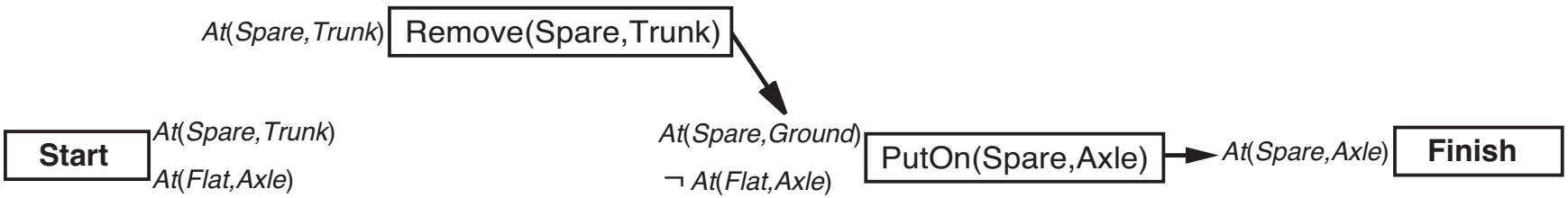
Plan-space search

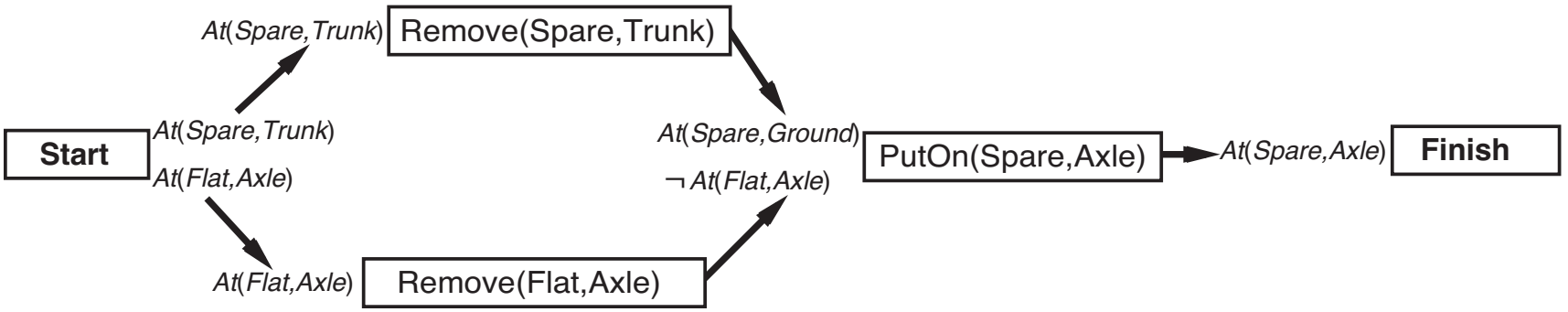


Plan-space search



Plan-space search

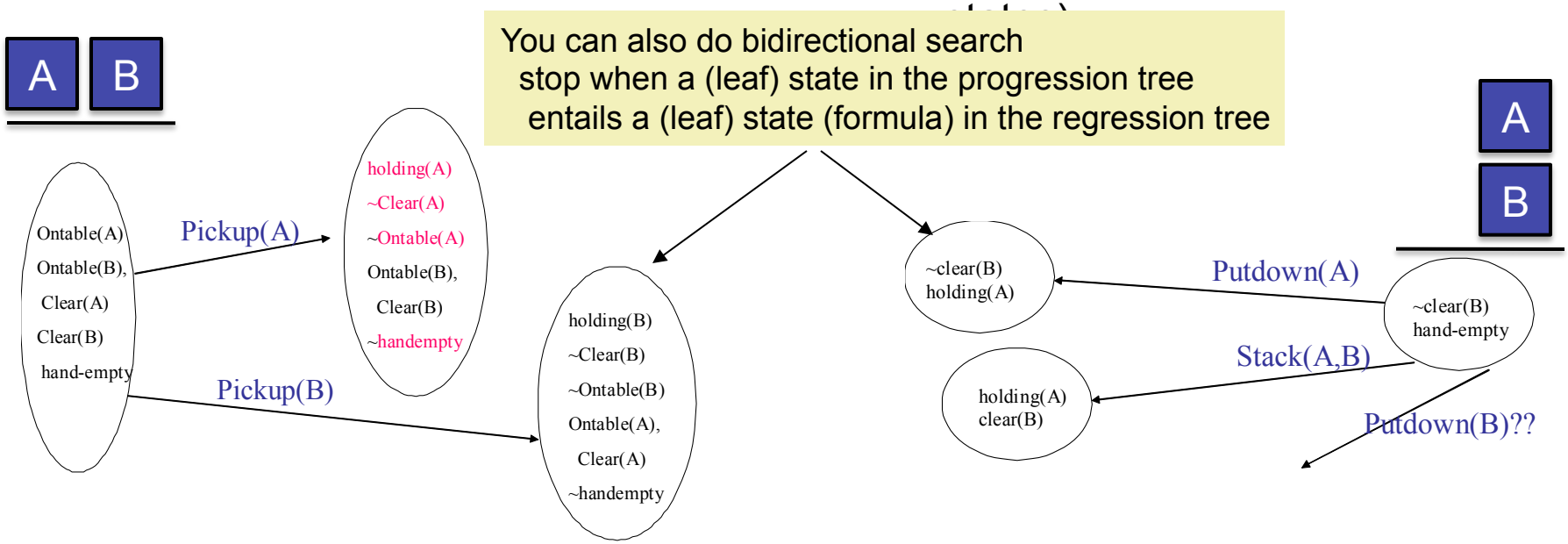




Progression vs. Regression

- Progression has higher branching factor
- Progression searches in the space of complete (and consistent) states

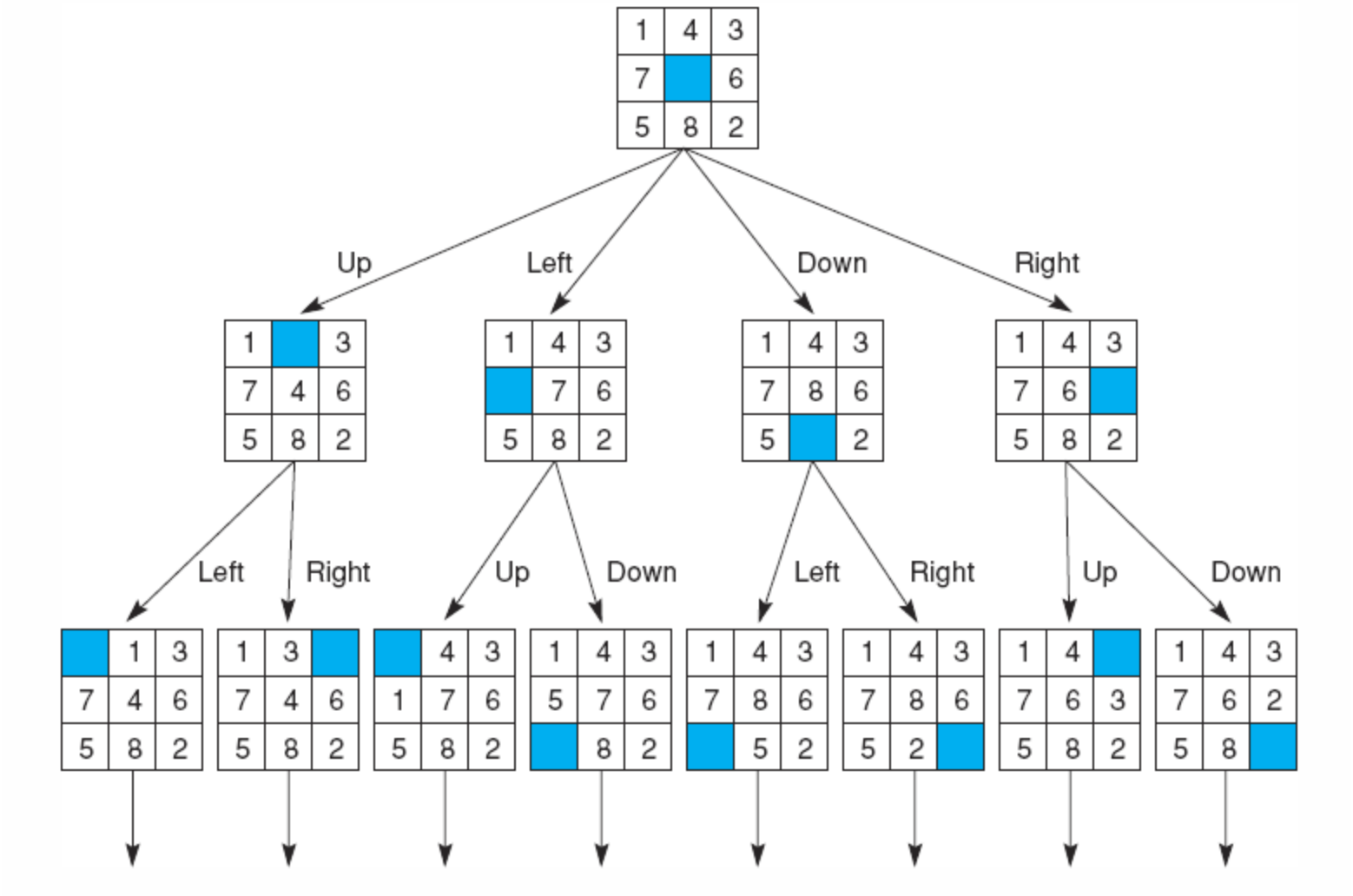
- Regression has lower branching factor
- Regression searches in the space of *partial states*
  - There are 3^n partial states (as against 2^n complete



State of the art

- Annual planning competitions
- Best technique has varied over time
- Currently: mostly forward state-space
- Largely due to good heuristics (relaxed prob.)
  - Heuristics for atomic (state search) problem
    Can only come from *outside* analysis of domain
  - Heuristics for factored (planning) problem
    Can be *domain-independent*

8-puzzle state space



8-puzzle action schema

- *Action*(*Slide*(*t, a, b*),

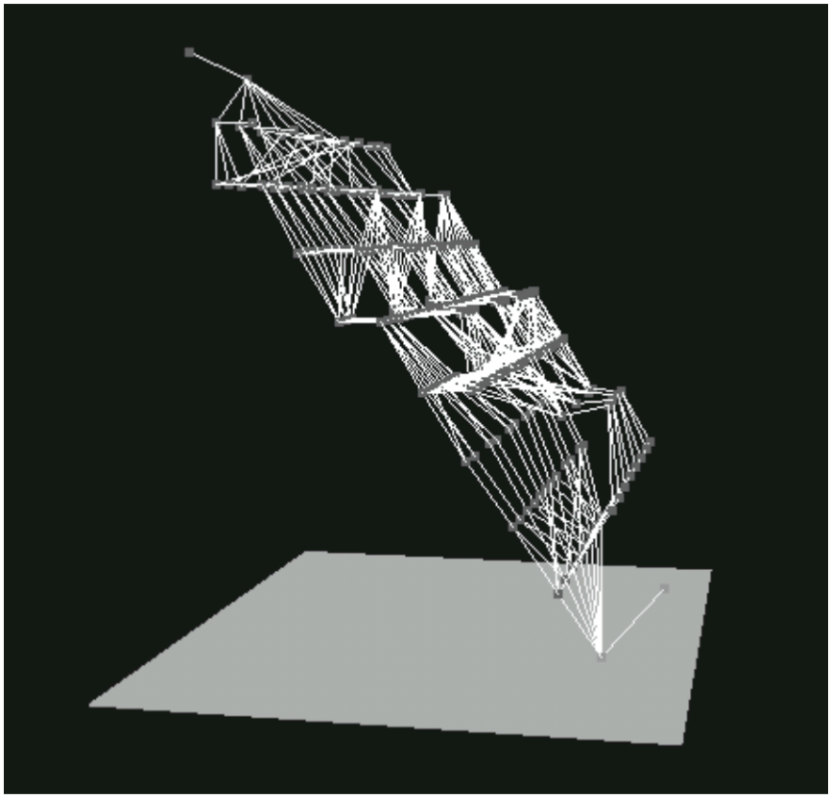
  Pre: *On*(*t, a*)∧ *Tile*(*t*) ∧ *Blank*(*b*) ∧ *Adjacent*(*a,b*)

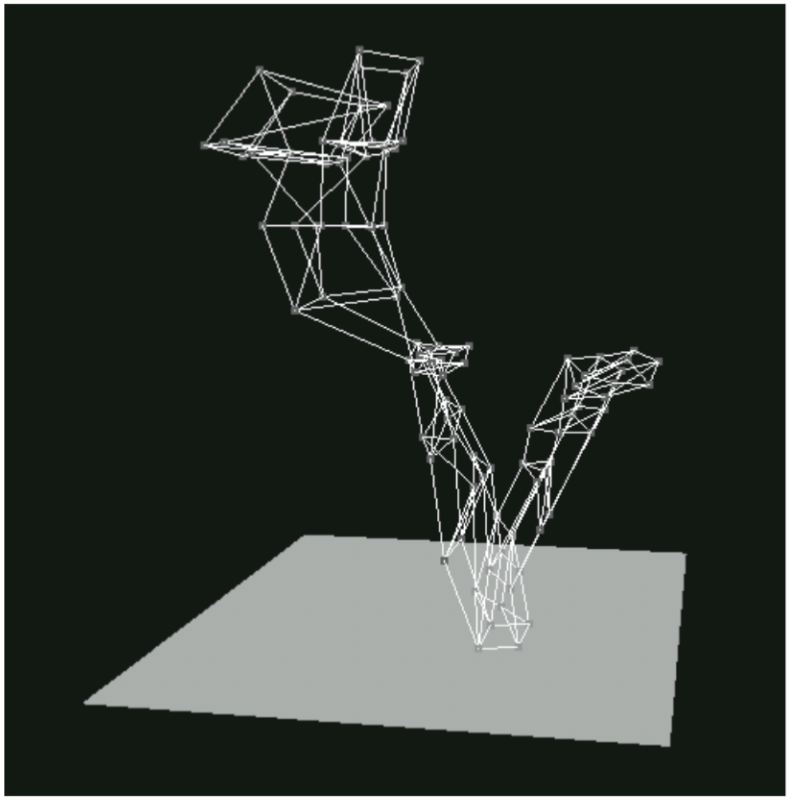
  Eff: *On*(t, b) ∧*Blank*(*a*) ∧ ¬*On*(t, a) ∧¬*Blank*(*b*))

8-puzzle heuristics

- *Action*(*Slide*(*t, a, b*),
  Pre: *On*(*t, a*) ∧ *Tile*(*t*) ∧ *Blank*(*b*) ∧ *Adjacent*(*a,b*)
  Eff: *On*(t, b) ∧ *Blank*(*a*) ∧ ¬*On*(t, a) ∧ ¬*Blank*(*b*))
- Relaxed problems: ignore preconditions
  - ~~*Blank*(*b*)~~:
  - Manhatten distance heuristic
  - ~~*Blank*(*b*) ∧ *Adjacent*(*a,b*)~~:
  - Number-of-misplaced-tiles heuristic
- Relaxed problems: ignore delete lists
  - Ignore negative effects
    Convex space: Find solution (non-optimal) by hill-climbing

Convex search: ignore del lists





Factored Rep allows control

Is there any Need for Domain-Dependent Control Information?

Matthew L. Ginsberg* and **Donald F. Geddis†**
Computer Science Department
Stanford University
Stanford, California 94305
ginsberg@cs.stanford.edu

Factored Rep allows control

Is there any Need for Domain-Dependent Control Information?

Matthew L. Ginsberg* and **Donald F. Geddis†**
Computer Science Department
Stanford University
Stanford, California 94305
ginsberg@cs.stanford.edu

Abstract

No.

1 Introduction

The split between *base-level* and *metalevel* knowledge has long been recognized by the declarative community. Roughly speaking, base-level knowledge has to do with information about some particular domain, while metalevel knowledge has to do with knowledge *about* that information. A typical base-level fact might be, "Iraq invaded Kuwait," while a typical metalevel fact might be, "To show that a country c is aggressive, first try to find another country that has been invaded by c."

because sentences expressing modal knowledge typically involve the use of predicate symbols that have other sentences as arguments. Thus a typical modal sentence might be, "I know that Iraq invaded Kuwait," or "I don't know of anyone that Kuwait has invaded." Note that the information here doesn't refer to the domain so much as it does to our knowledge *about* the domain; nor is it control information telling us what to do with this knowledge. It simply reports on the state of our information at some particular point in time. In general, we will describe as modal all information describing the structure of our declarative knowledge in some way.

The claim we are making – and we intend to prove it – is that there is no place in declarative systems

Beyond Classical Planning

- Convenient to have more expressive lang. "Move *all* the cargo from SFO to JFK"
- Can be done with careful extensions to factored planning language
- *Or:* Use existing first-order logical provers
- Strong foundation for studying planning
- Still, less used in practice than other techniques

First-Order Logic

- And, Or, Not, Implies
  (as in propositional logic)
- Variables ranging over *objects*
- Relations and functions over objects
- Quantifiers ∀ (for all) and ∃ (exists)
  - Goal: ∀ c Cargo(c) ⇒ At(c, JFK)

Situation Calculus

- Actions are objects
- Situations are objects
- Function: s2 = Result(s, a)
- Fluents: At(C1, JFK, s) *change over time*
- Possibility axioms
  - Say when an action is possible
- Successor-state axioms
  - Partially describe the resulting state of an action

Situation Calculus

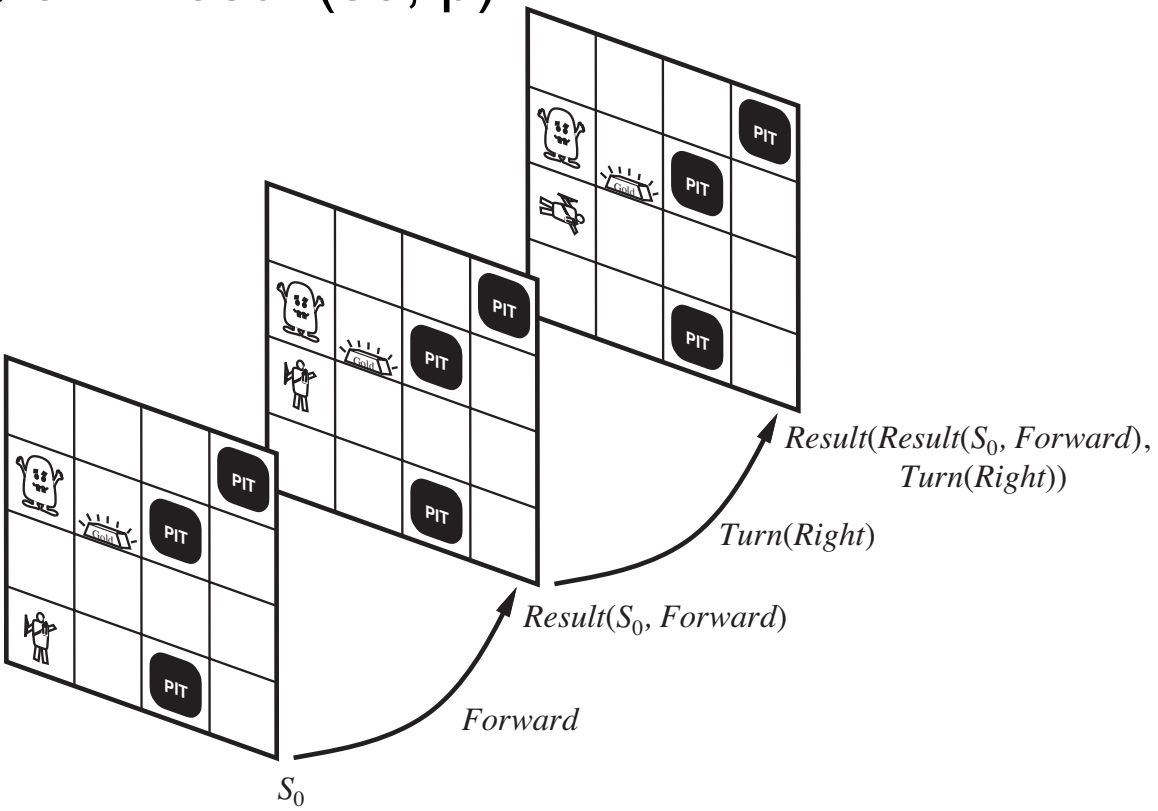
- Possibility Axioms (for each action)
  - $SomeFormula(s) \Rightarrow Poss(a, s)$
  - $Alive(Agent, s) \wedge Have(Agent, Arrow, s) \Rightarrow Poss(Shoot, s)$
- Successor-state Axiom (for each fluent)
  - $Poss(a, s) \Rightarrow$ (*fluent is true* $\Leftrightarrow$ *a made it true* $\vee$ *it was true and a left it alone*)
  - $Poss(a, s) \Rightarrow (Holding(Agent, g, Result(s, a)) \Leftrightarrow a = Grab(g) \vee (Holding(Agent, g, s) \wedge a \neq Release(g)))$

Situations as Result of Action

$\exists s, p : Goal(s) \wedge s = Result(s0, p)$

Situation Calculus

First-order Logic



s = Result(s, [])

Result(s, [a,b,…]) = Result(Result(s, a), [b,…])

Planning Graphs

- Planning graphs are an efficient way to create a representation of a planning problem that can be used to
  - Achieve better heuristic estimates
  - Directly construct plans
- Planning graphs only work for propositional problems
  - Compile to propositional if necessary

Planning Graphs

- Planning graphs consists of a seq of levels that correspond to time steps in the plan.
  - Level 0 is the initial state.
  - Each level consists of a set of literals and a set of actions that represent what *might be* possible at that step in the plan
  - *Might be* is the key to efficiency
  - Records only a restricted subset of possible negative interactions among actions.

Planning Graphs

- Each level consists of
- ***Literals*** = all those that *could* be true at that time step, depending upon the actions executed at preceding time steps.
- ***Actions*** = all those actions that *could* have their preconditions satisfied at that time step, depending on which of the literals actually hold.

Planning Graph Example

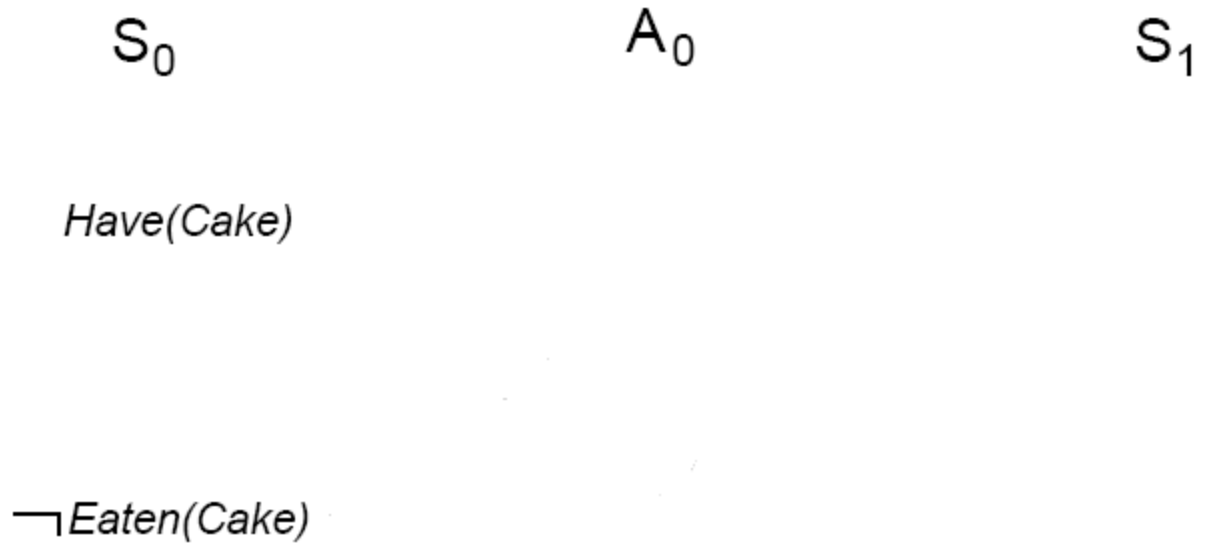
Init(Have(Cake))

Goal(Have(Cake) ∧ Eaten(Cake))

Action(Eat(Cake),
  PRECOND: Have(Cake)
  EFFECT: ¬Have(Cake) ∧ Eaten(Cake))

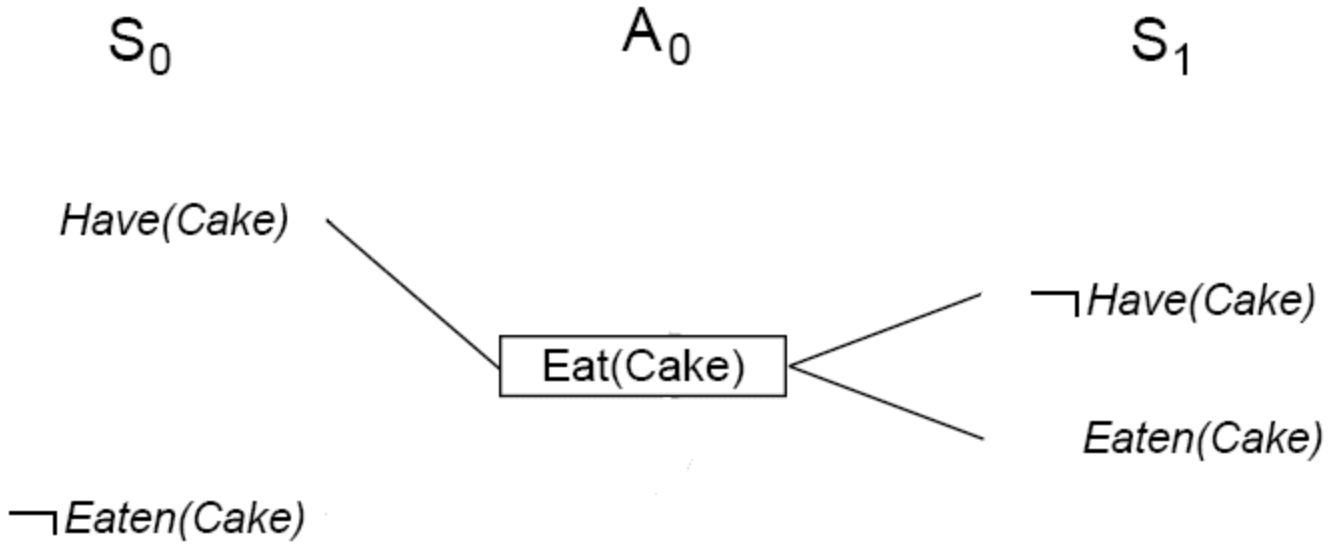
Action(Bake(Cake),
  PRECOND: ¬ Have(Cake)
  EFFECT: Have(Cake))

Planning Graph Example



Create level 0 from initial problem state.

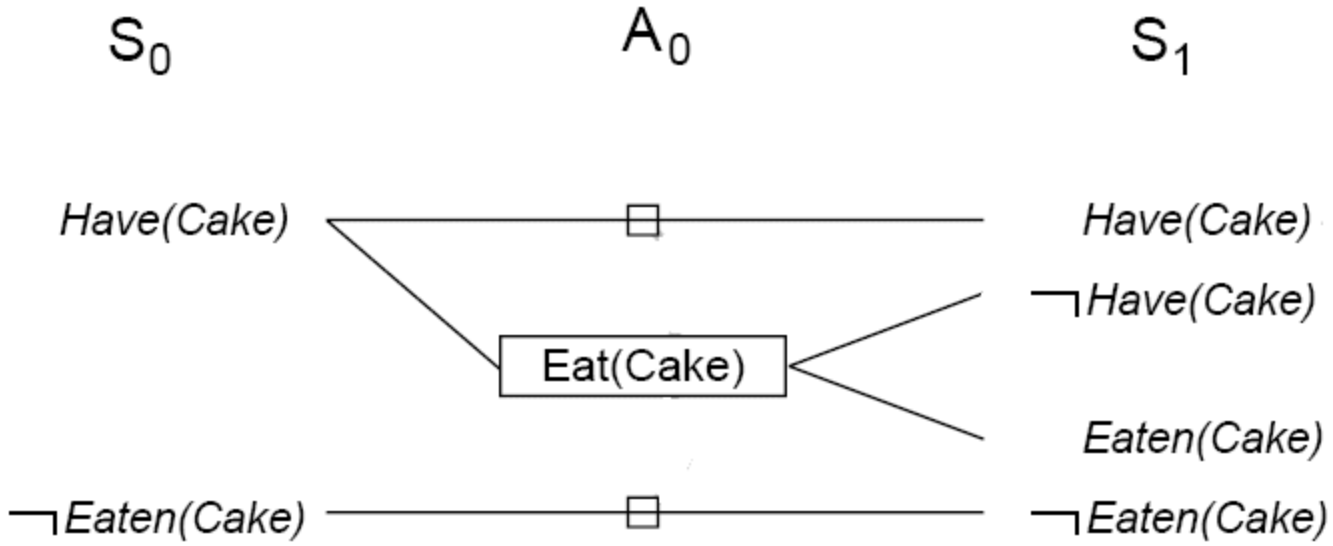
Planning Graph Example



Add all applicable actions.

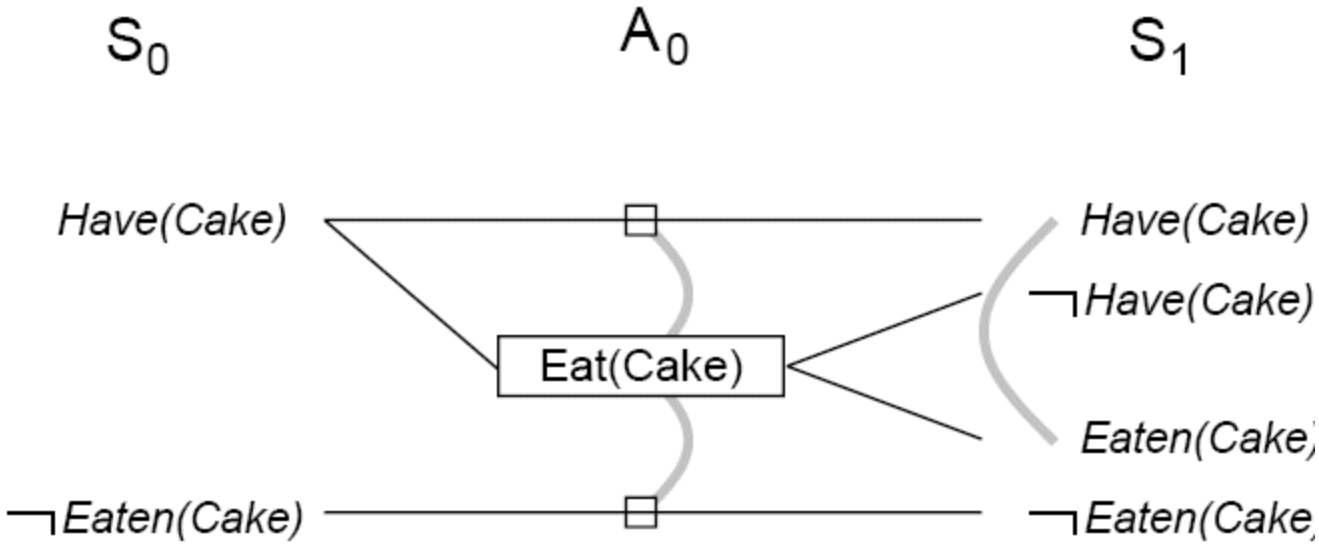
Add all effects to the next state.

Planning Graph Example



Add *persistence actions* (inaction = no-ops) to map all literals in state S_i to state S_{i+1}.

Planning Graph Example

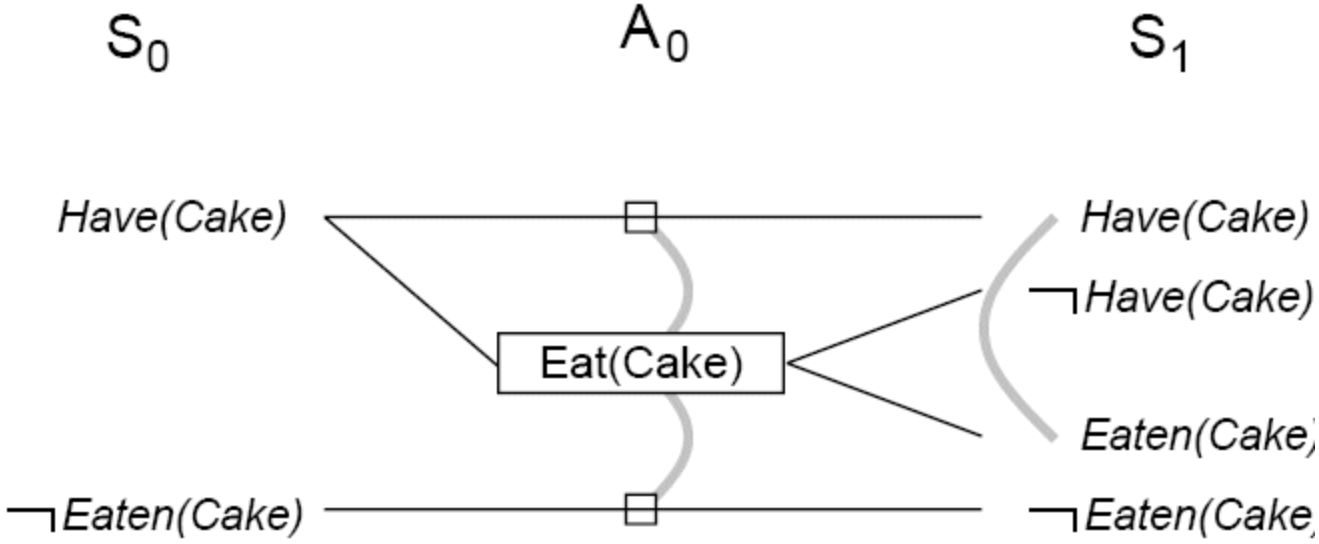


Identify *mutual exclusions* between actions and literals based on potential conflicts.

Mutual exclusion

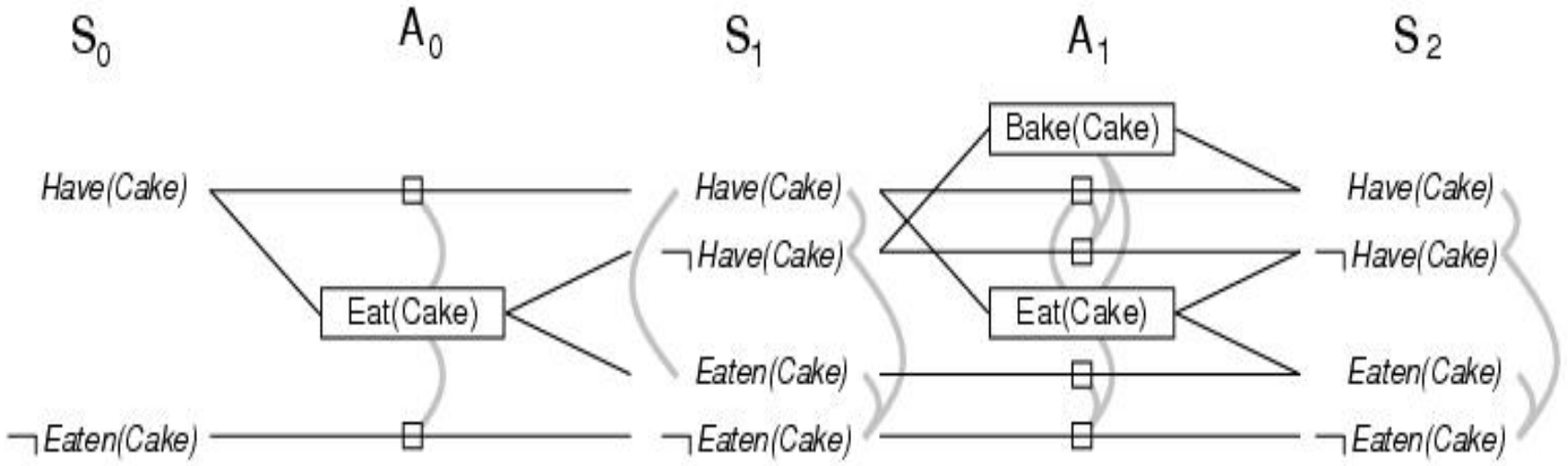
- A mutex relation holds between **two actions** when:
  - *Inconsistent effects*: one action negates the effect of another.
  - *Interference*: one of the effects of one action is the negation of a precondition of the other.
  - *Competing needs*: one of the preconditions of one action is mutually exclusive with the precondition of the other.
- A mutex relation holds between **two literals** when:
  - one is the negation of the other OR
  - each possible action pair that could achieve the literals is mutex (inconsistent support).

Cake example



- Level S_1 contains all literals that could result from picking any subset of actions in A_0
  - Conflicts between literals that can not occur together (as a consequence of the selection action) are represented by mutex links.
  - S1 defines multiple states and the mutex links are the constraints that define this set of states.

Cake example



- Repeat process until graph levels off:
  - two consecutive levels are identical

PG and Heuristic Estimation

- PG's provide information about the problem
  - PG is a relaxed problem.
  - A literal that does not appear in the final level of the graph cannot be achieved by any plan.
    - $h(s) = \infty$
  - Level Cost: First level in which a goal appears
    - Very low estimate, since several actions can occur
    - Improvement: restrict to one action per level using *serial PG* (add mutex links between *every* pair of actions, except persistence actions).

PG and Heuristic Estimation

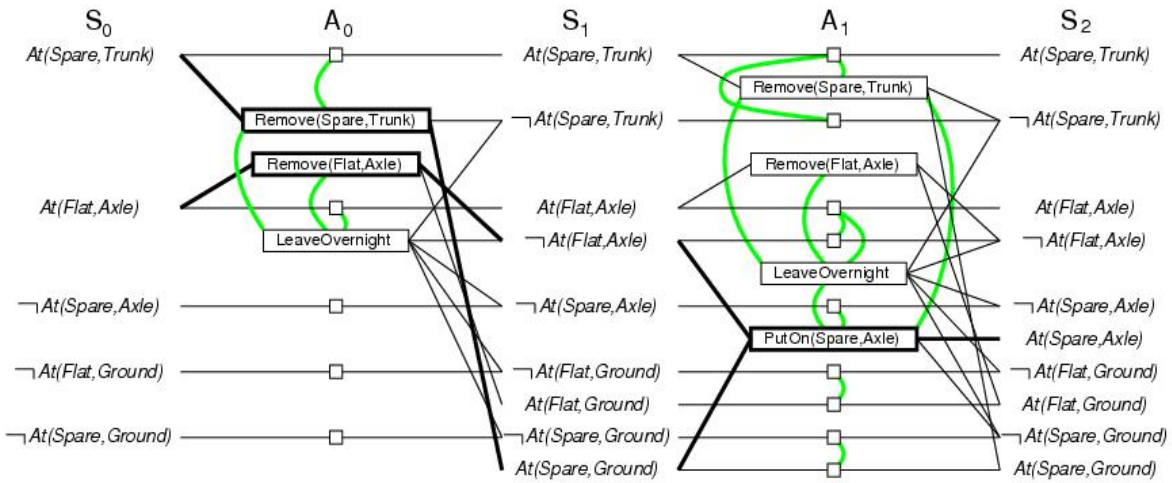
- Cost of a conjunction of goals
  - Max-level: maximum first level of any of the goals
  - Sum-level: sum of first levels of all the goals
  - Set-level: First level in which all goals appear without being mutex

The GRAPHPLAN Algorithm

- Extract a solution directly from the PG

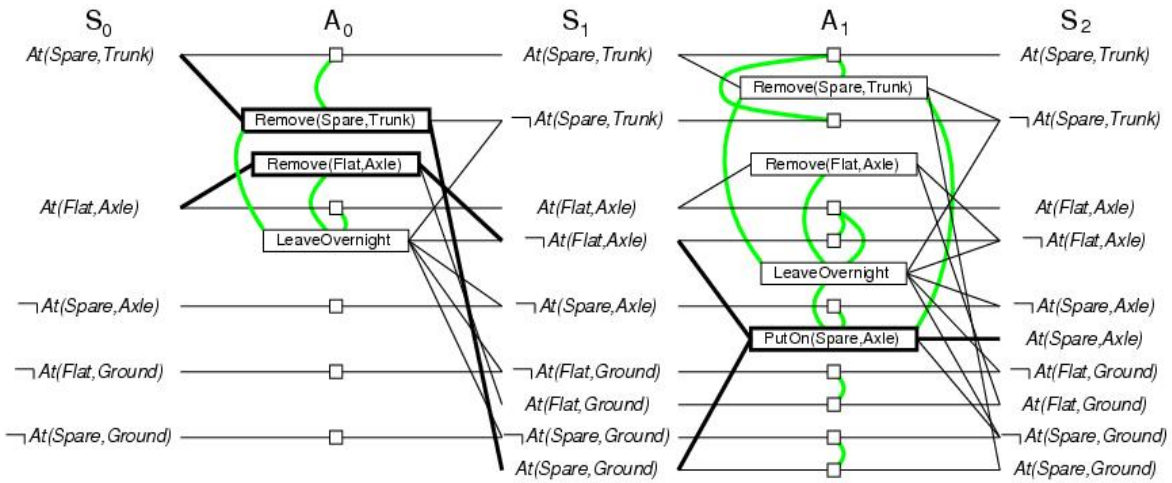
function GRAPHPLAN(*problem*) **return** *solution* or failure
    graph ← INITIAL-PLANNING-GRAPH(*problem*)
    goals ← GOALS[*problem*]
    loop do
        if *goals* all non-mutex in last level of graph **then do**
            solution ← EXTRACT-SOLUTION(*graph, goals,* LEN*(graph)*)
            if *solution* ≠ failure **then return** *solution*
            else if NO-SOLUTION-POSSIBLE(*graph*) **then return** failure
        graph ← EXPAND-GRAPH(*graph, problem*)

GRAPHPLAN example



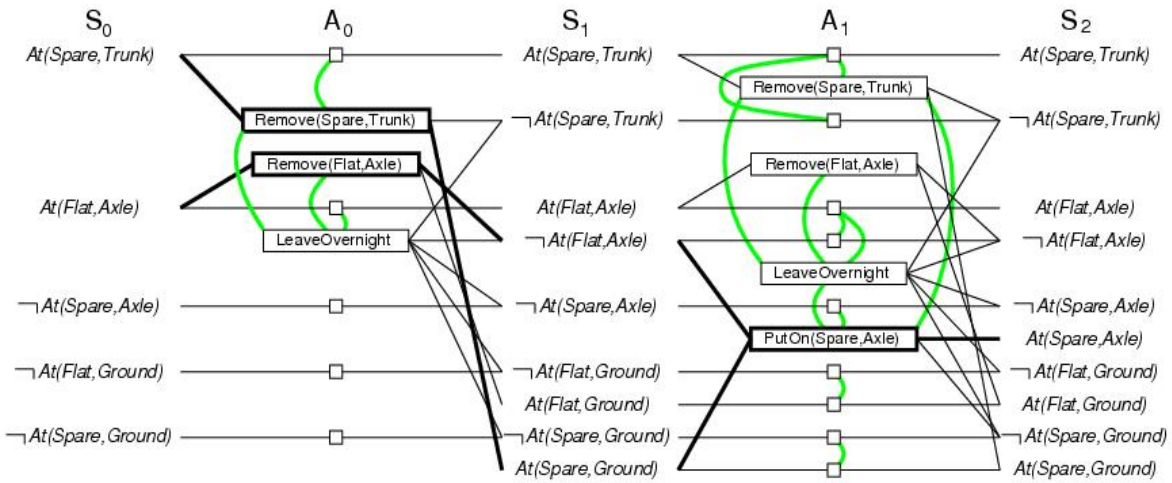
- Initially the plan consist of 5 literals from the initial state (S0).
- Add actions whose preconditions are satisfied by EXPAND-GRAPH (A0)
- Also add persistence actions and mutex relations.
- Add the effects at level S1
- Repeat until goal is in level Si

GRAPHPLAN example



- EXPAND-GRAPH also looks for mutex relations
  - Inconsistent effects
    - E.g. Remove(Spare, Trunk) and LeaveOverNight due to At(Spare,Ground) and **not** At(Spare, Ground)
  - Interference
    - E.g. Remove(Flat, Axle) and LeaveOverNight At(Flat, Axle) as PRECOND and **not** At(Flat,Axle) as EFFECT
  - Competing needs
    - E.g. PutOn(Spare,Axle) and Remove(Flat, Axle) due to At(Flat.Axle) and **not** At(Flat, Axle)
  - Inconsistent support
    - E.g. in S2, At(Spare,Axle) and At(Flat,Axle)

GRAPHPLAN example



- In S2, the goal literals exist and are not mutex with any other
  - Solution might exist and EXTRACT-SOLUTION will try to find it
- EXTRACT-SOLUTION can search with:
  - Initial state = last level of PG and goal goals of planning problem
  - Actions = select any set of non-conflicting actions that cover the goals in the state
  - Goal = reach level S0 such that all goals are satisfied
  - Cost = 1 for each action.

GRAPHPLAN Termination

- Termination of graph construction? YES
- PG are monotonically increasing or decreasing:
  - Literals increase monotonically
  - Actions increase monotonically
  - Mutexes decrease monotonically
- Because of these properties and because there is a finite number of actions and literals, every PG will eventually level off