Action Planning

Where logic-based representation of knowledge makes search problems more interesting

In which we see how an agent can take advantage of the structure of a problem to efficiently construct complex plans of action

R&N: Chap. 11, Sect. 11.1–4

Slides from Jean-Claude Latombe at Stanford University
(used with permission)

Summary

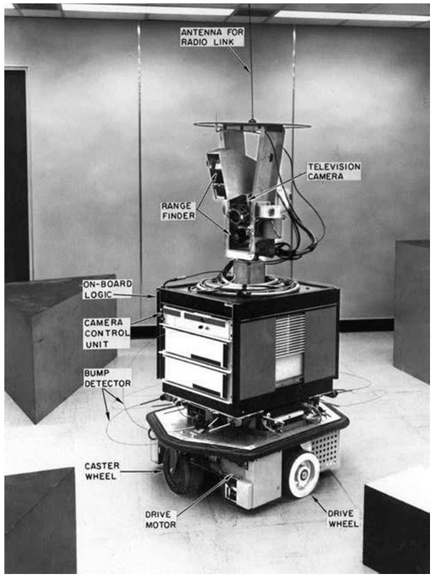
- The goal of action planning is to choose actions and ordering relations among these actions to achieve specified goals
- Search-based problem solving applied to 8-puzzle was one example of planning, but our description of this problem used specific data structures and functions
- Here, we will develop a non-specific, logic-based language to represent knowledge about actions, states, and goals, and we will study how search algorithms can exploit this representation

Some Applications of AI Planning

- Military operations
- Operations in container ports
- Construction tasks
- Machining and manufacturing
- Autonomous control of satellites and other spacecrafts

Knowledge Representation Tradeoff

- Expressiveness vs. computational efficiency
- STRIPS: a simple, still reasonably expressive planning language based on propositional logic
  1. Examples of planning problems in STRIPS
  2. Planning methods
  3. Extensions of STRIPS



SHAKEY the robot

- Like programming, knowledge representation is still an art

Part I

STRIPS LANGUAGE THROUGH EXAMPLES

Stanford Research Institute Problem Solver (STRIPS)

- Richard Fikes and Nils Nilsson in 1971
- **Action language:** base for most automatic planning problems

Stanford Research Institute Problem Solver (STRIPS)

$\langle P, O, I, G \rangle$

P set to **conditions (propositions)**

O set of **operators (actions):**

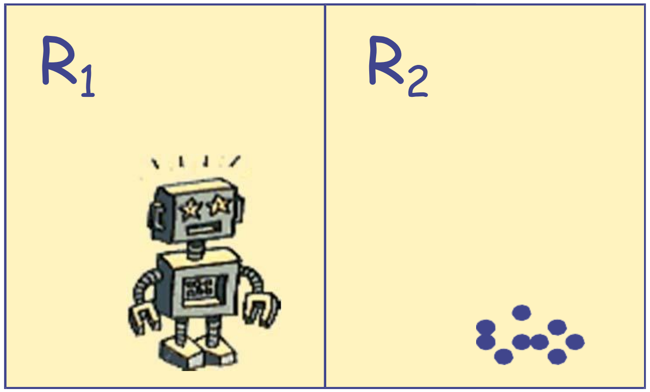
- $\langle \alpha, \beta, \gamma, \delta \rangle$ each operator
  - α: true conditions for the action
  - β: false conditions for the action
  - γ: conditions made true by the action
  - δ: conditions made false by the action

I is the **initial state**

G is the **goal state:**

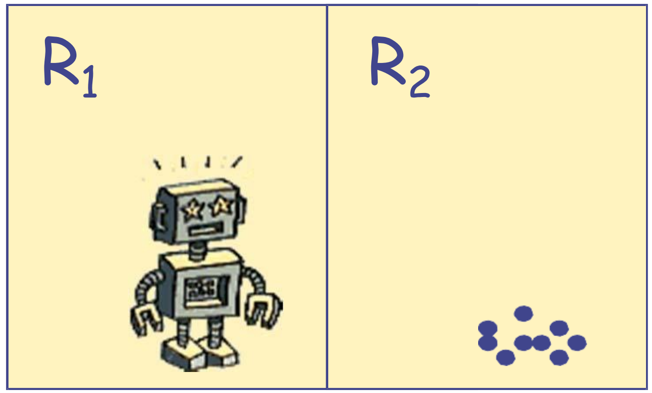
- $\langle N, M \rangle$: true and false conditions for the goal state

Vacuum-Robot Example



- Two rooms: R_1 and R_2
- A vacuum robot
- Dust

State Representation

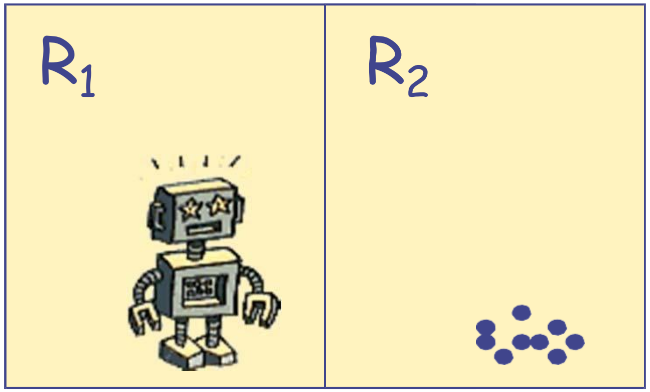


$$\mathrm{In}(\mathrm{Robot}, R_1) \wedge \mathrm{Clean}(R_1)$$

Propositions that “hold” (i.e. are true) in the state

Logical “and” connective

State Representation



$$In(Robot, R_1) \wedge Clean(R_1)$$

- Conjunction of propositions
- No negated proposition, such as $\neg Clean(R_2)$
- Closed-world assumption: Every proposition that is not listed in a state is false in that state
- No "or" connective, such as $In(Robot,R_1) \vee In(Robot,R_2)$
- No variable, e.g., $\exists x\, Clean(x)$

Goal Representation

Example: $Clean(R_1) \wedge Clean(R_2)$

- Conjunction of propositions
- No negated proposition
- No “or” connective
- No variable

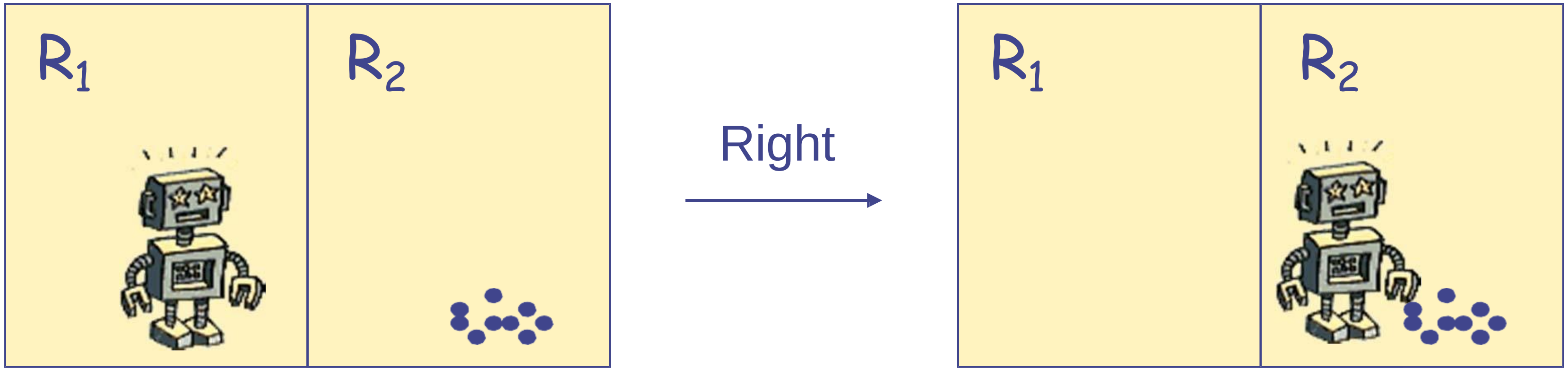
A goal G is achieved in a state S:

if all the propositions in G (sub-goals) are also in S

Action Representation

Right

- Precondition = In(Robot, R_1)
- Delete-list = In(Robot, R_1)
- Add-list = In(Robot, R_2)



In(Robot, R_1) $\wedge$ Clean(R_1)

In(Robot, R_2) $\wedge$ Clean(R_1)

Action Representation

Right

- Precondition = In(Robot, R_1)
- Delete-list = In(Robot, R_1)
- Add-list = In(Robot, R_2)

Sets of propositions

Same form as a goal: conjunction of propositions

Action Representation

Right

- Precondition = In(Robot, R_1)
- Delete-list = In(Robot, R_1)
- Add-list = In(Robot, R_2)

- An action **A** is applicable to a state **S** if the propositions in its precondition are all in **S**
- The application of **A** to **S** is a new state obtained by deleting the propositions in the delete list from **S** and adding those in the add list

Other Actions

Left

- $P = In(Robot, R_2)$
- $D = In(Robot, R_2)$
- $A = In(Robot, R_1)$

Suck(R_1)

- $P = In(Robot, R_1)$
- $D = \emptyset$ [empty list]
- $A = Clean(R_1)$

Suck(R_2)

- $P = In(Robot, R_2)$
- $D = \emptyset$ [empty list]
- $A = Clean(R_2)$

Other Actions

Left

- P = In(Robot, R_2)
- D = In(Robot, R_2)
- A = In(Robot, R_1)

Suck(r)

- P = In(Robot, r)
- D = ∅ [empty list]
- A = Clean(r)

Action Schema

Describe several actions, here:

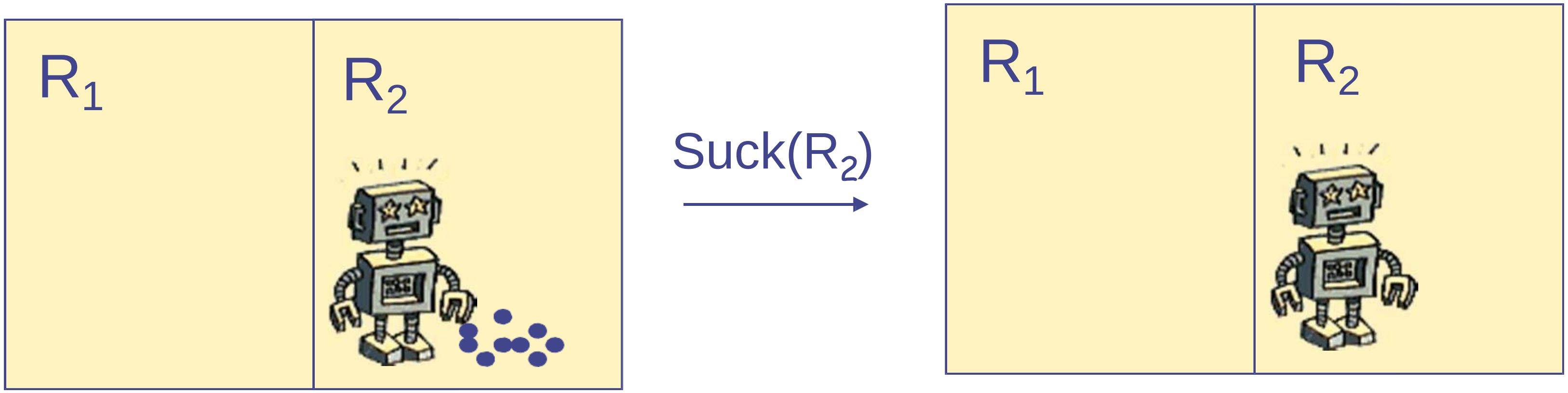
Suck(R_1) and **Suck(R_2)**

Parameter that gets "instantiated" by matching the precondition against a state

Suck(r)

- P = In(Robot, r)
- D = ∅ [empty list]
- A = Clean(r)

Action Schema



$In(Robot, R_2) \wedge Clean(R_1)$

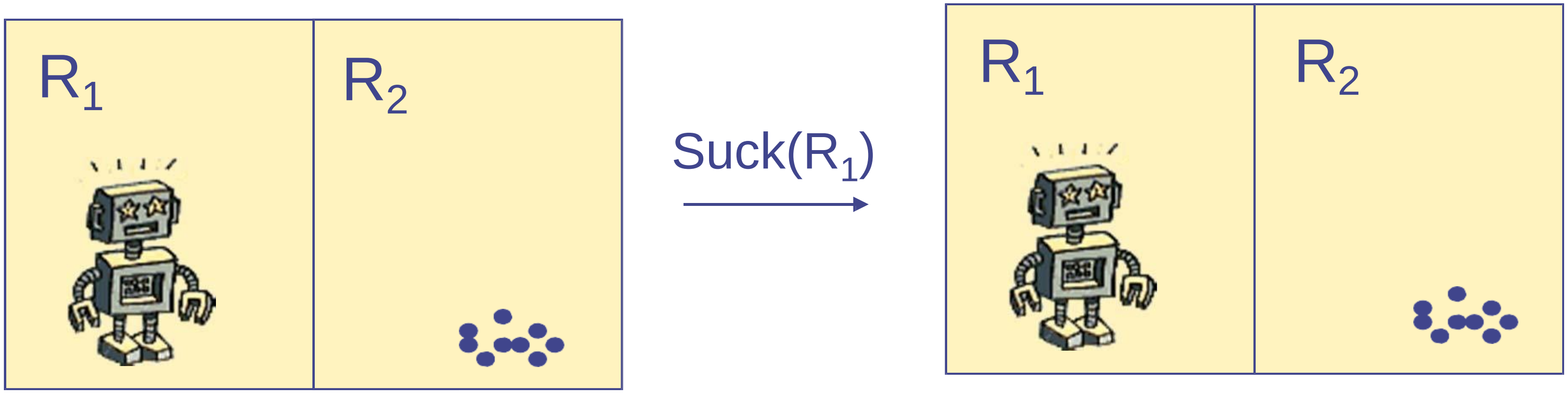
$In(Robot, R_2) \wedge Clean(R_1)$
$\wedge\ Clean(R_2)$

$r \leftarrow R_2$

Suck(r)

- P = In(Robot, r)
- D = ∅
- A = Clean(r)

Action Schema



In(Robot, R_1) $\wedge$ Clean(R_1)

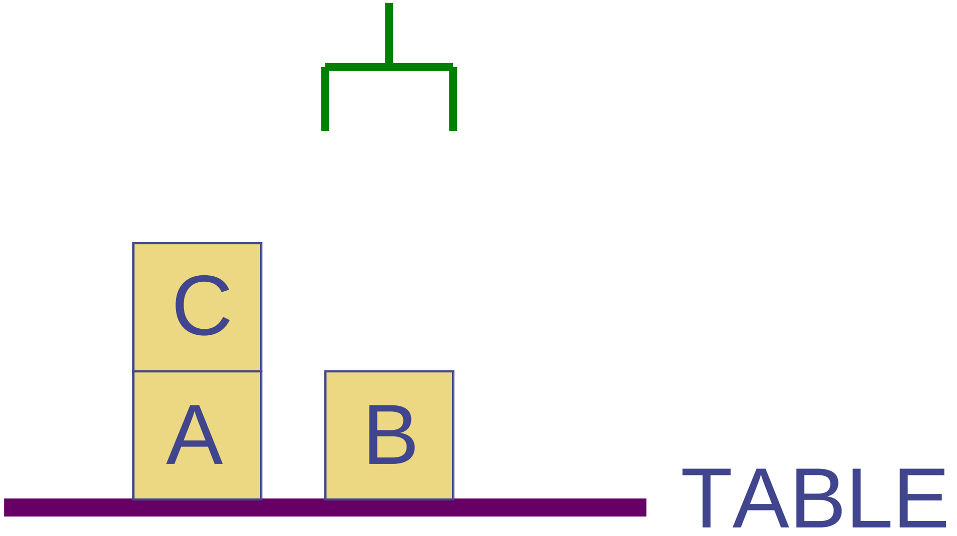
In(Robot, R_1) $\wedge$ Clean(R_1)

$r \leftarrow R_1$

Suck(r)

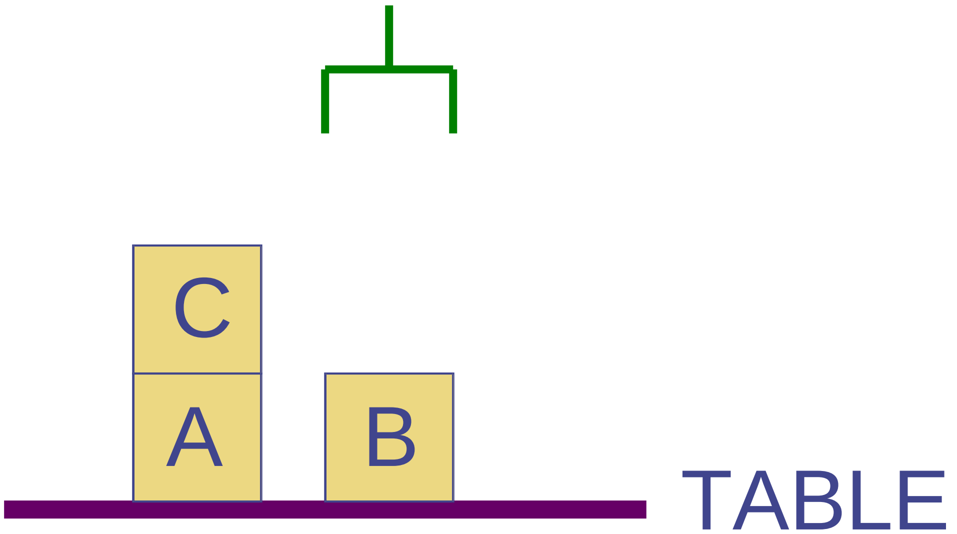
- P = In(Robot, r)
- D = ∅
- A = Clean(r)

Blocks-World Example



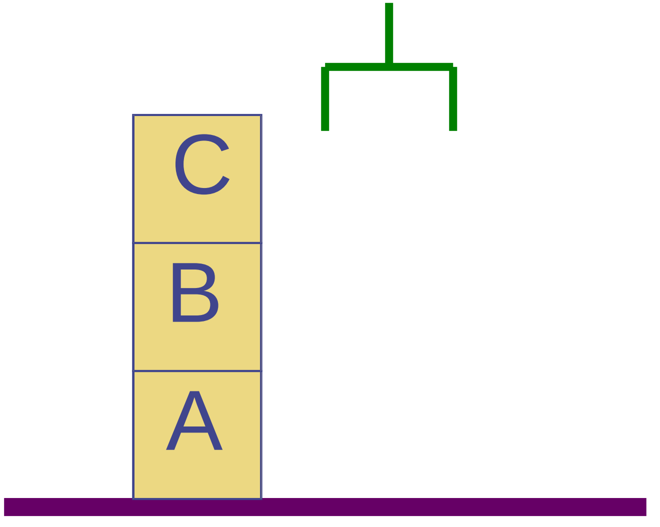
- A robot hand can move blocks on a table
- The hand cannot hold more than one block at a time
- No two blocks can fit directly on the same block
- The table is arbitrarily large

State



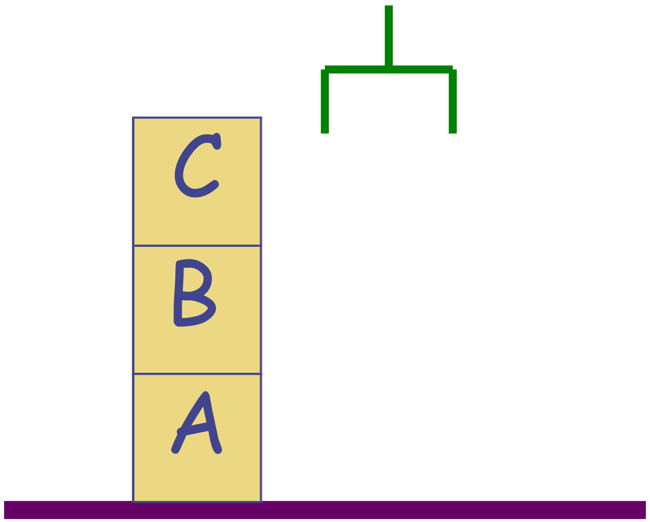
Block(A) ∧ Block(B) ∧ Block(C) ∧ On(A,TABLE) ∧ On(B,TABLE) ∧ On(C,A) ∧ Clear(B) ∧ Clear(C) ∧ Handempty

Goal



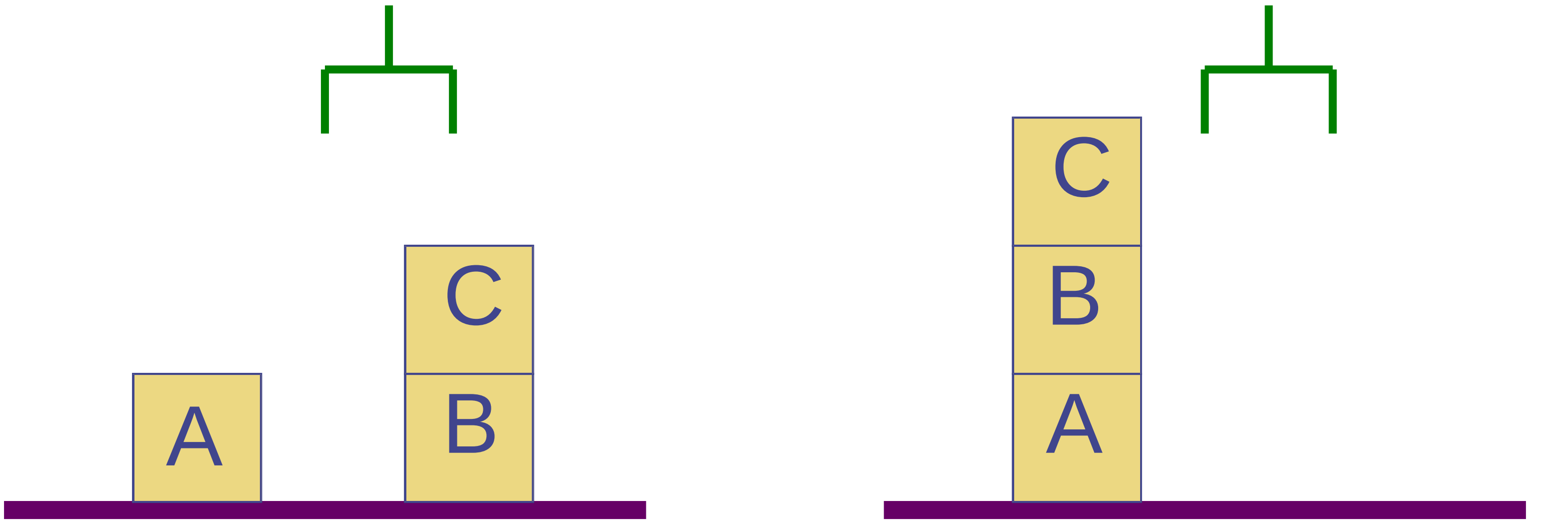
On(A,TABLE) ∧ On(B,A) ∧ On(C,B) ∧ Clear(C)

Goal



On(A,TABLE) ∧On(B,A) ∧On(C,B) ∧Clear(C)

Goal



On(A,TABLE) ∧ On(C,B)

Action

Unstack(x,y)

P = Handempty ∧ Block(x) ∧ Block(y) ∧ Clear(x) ∧ On(x,y)

D = Handempty, Clear(x), On(x,y)

A = Holding(x), Clear(y)

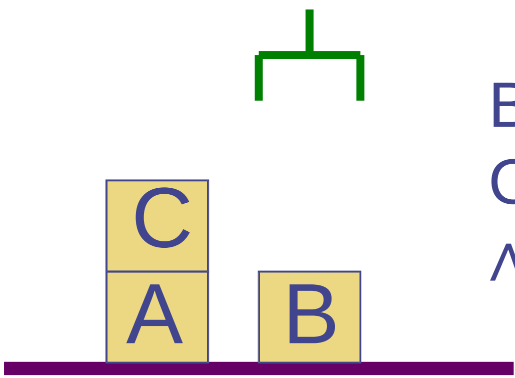
Action

Unstack(x,y)

P = Handempty ∧ Block(x) ∧ Block(y) ∧ Clear(x) ∧ On(x,y)

D = Handempty, Clear(x), On(x,y)

A = Holding(x), Clear(y)



Block(A) ∧ Block(B) ∧ Block(C) ∧
On(A,TABLE) ∧ On(B,TABLE) ∧ On(C,A)
∧ Clear(B) ∧ Clear(C) ∧ Handempty

Unstack(C,A)

P = Handempty ∧ Block(C) ∧ Block(A) ∧ Clear(C) ∧ On(C,A)

D = Handempty, Clear(C), On(C,A)

A = Holding(C), Clear(A)

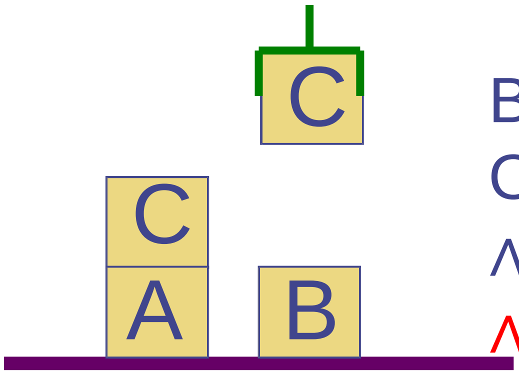
Action

Unstack(x,y)

P = Handempty ∧ Block(x) ∧ Block(y) ∧ Clear(x) ∧ On(x,y)

D = Handempty, Clear(x), On(x,y)

A = Holding(x), Clear(y)



Block(A) ∧ Block(B) ∧ Block(C) ∧ On(A,TABLE) ∧ On(B,TABLE) ∧ ~~On(C,A)~~ ∧ Clear(B) ∧ ~~Clear(C)~~ ∧ ~~Handempty~~ ∧ Holding(C) ∧ Clear(A)

Unstack(C,A)

P = Handempty ∧ Block(C) ∧ Block(A) ∧ Clear(C) ∧ On(C,A)

D = Handempty, Clear(C), On(C,A)

A = Holding(C), Clear(A)

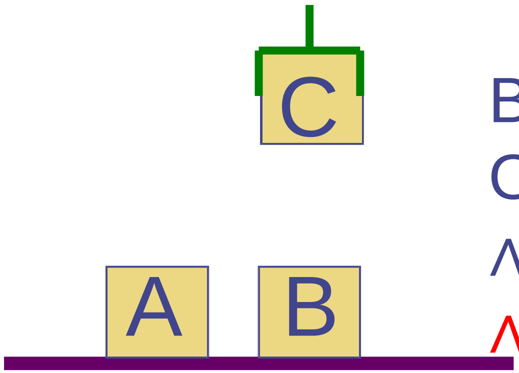
Action

Unstack(x,y)

P = Handempty ∧ Block(x) ∧ Block(y) ∧ Clear(x) ∧ On(x,y)

D = Handempty, Clear(x), On(x,y)

A = Holding(x), Clear(y)



Block(A) ∧ Block(B) ∧ Block(C) ∧
On(A,TABLE) ∧ On(B,TABLE) ∧ ~~On(C,A)~~
∧ Clear(B) ∧ ~~Clear(C)~~ ∧ ~~Handempty~~
∧ Holding(C) ∧ Clear(A)

Unstack(C,A)

P = Handempty ∧ Block(C) ∧ Block(A) ∧ Clear(C) ∧ On(C,A)

D = Handempty, Clear(C), On(C,A)

A = Holding(C), Clear(A)

All Actions

Unstack(x,y)

P = Handempty ∧ Block(x) ∧ Block(y) ∧ Clear(x) ∧ On(x,y)

D = Handempty, Clear(x), On(x,y)

A = Holding(x), Clear(y)

Stack(x,y)

P = Holding(x) ∧ Block(x) ∧ Block(y) ∧ Clear(y)

D = Clear(y), Holding(x)

A = On(x,y), Clear(x), Handempty

Pickup(x)

P = Handempty ∧ Block(x) ∧ Clear(x) ∧ On(x,Table)

D = Handempty, Clear(x), On(x,Table)

A = Holding(x)

Putdown(x)

P = Holding(x) ∧ Block(x)

D = Holding(x)

A = On(x,Table), Clear(x), Handempty

All Actions

Unstack(x,y)

P = Handempty ∧ Block(x) ∧ Block(y) ∧ Clear(x) ∧ On(x,y)

D = Handempty, Clear(x), On(x,y)

A = Holding(x), Clear(y)

Stack(x,y)

P = Holding(x) ∧ Block(x) ∧ Block(y) ∧ Clear(y)

D = Clear(y), Holding(x)

A = On(x,y), Clear(x), Handempty

Pickup(x)

P = Handempty ∧ Block(x) ∧ Clear(x) ∧ On(x,Table)

D = Handempty, Clear(x), On(x,Table)

A = Holding(x)

A block can always fit on the table

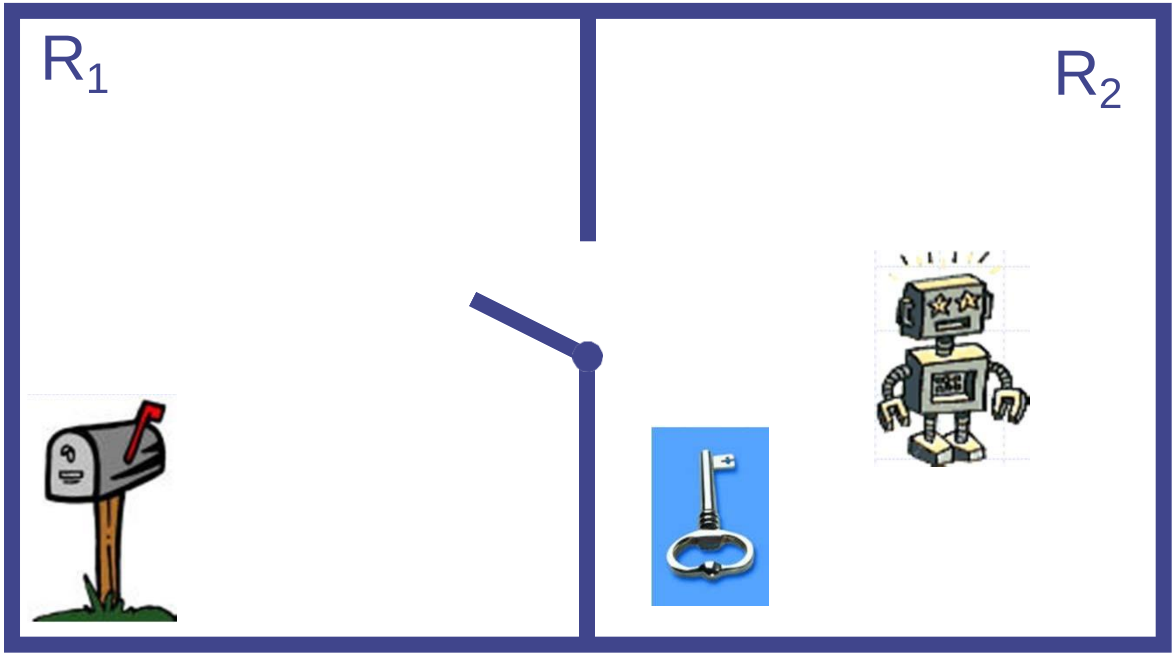
Putdown(x)

P = Holding(x) ∧ Block(x)

D = Holding(x)

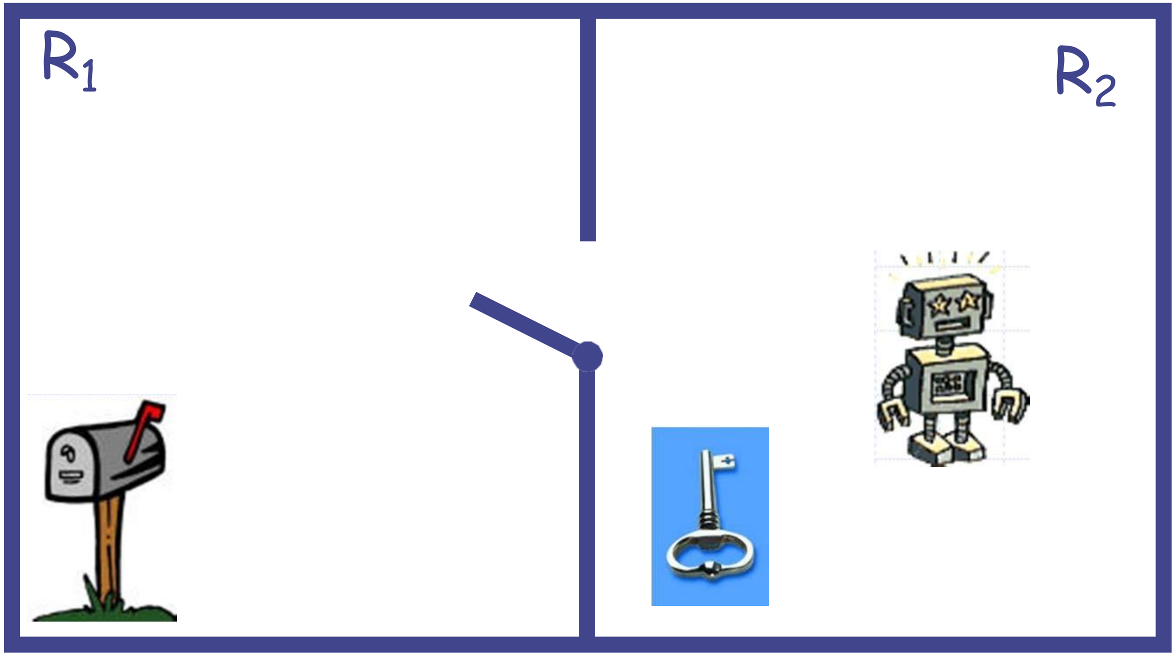
A = On(x,Table), Clear(x), Handempty

Key-in-Box Example



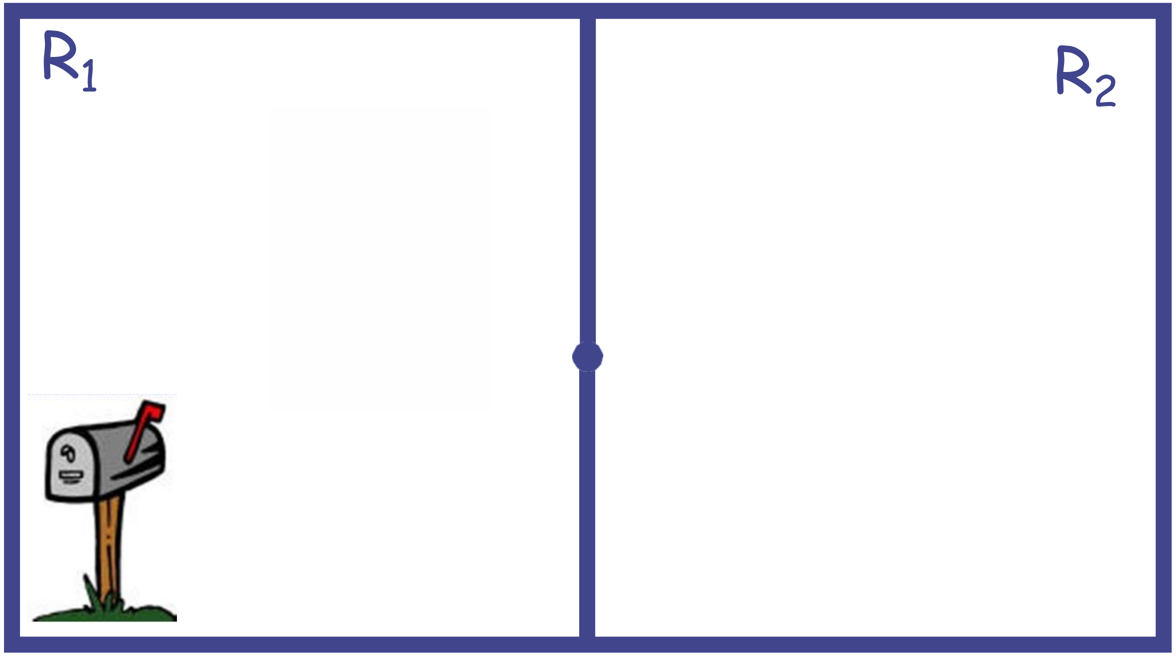
- The robot must lock the door and put the key in the box
- The key is needed to lock and unlock the door
- Once the key is in the box, the robot can't get it back

Initial State



In(Robot,R_2) ∧ In(Key,R_2) ∧ Unlocked(Door)

Goal



Locked(Door) ∧ In(Key,Box)

The robot's location isn't specified in the goal

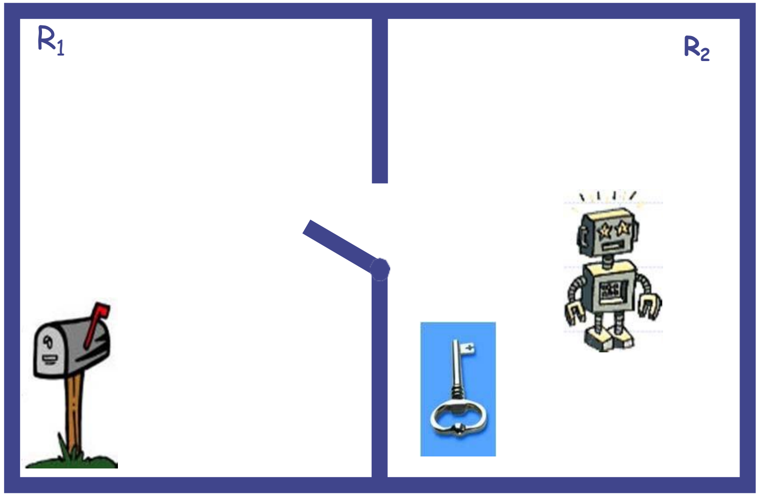
Actions

Grasp-Key-in-R_2

P = $In(Robot,R_2) \wedge In(Key,R_2)$

D = $\emptyset$

A = $Holding(Key)$



Lock-Door

P = $Holding(Key)$

D = $\emptyset$

A = $Locked(Door)$

Move-Key-from-R_2-into-R_1

P = $In(Robot,R_2) \wedge Holding(Key) \wedge Unlocked(Door)$

D = $In(Robot,R_2), In(Key,R_2)$

A = $In(Robot,R_1), In(Key,R_1)$

Put-Key-Into-Box

P = $In(Robot,R_1) \wedge Holding(Key)$

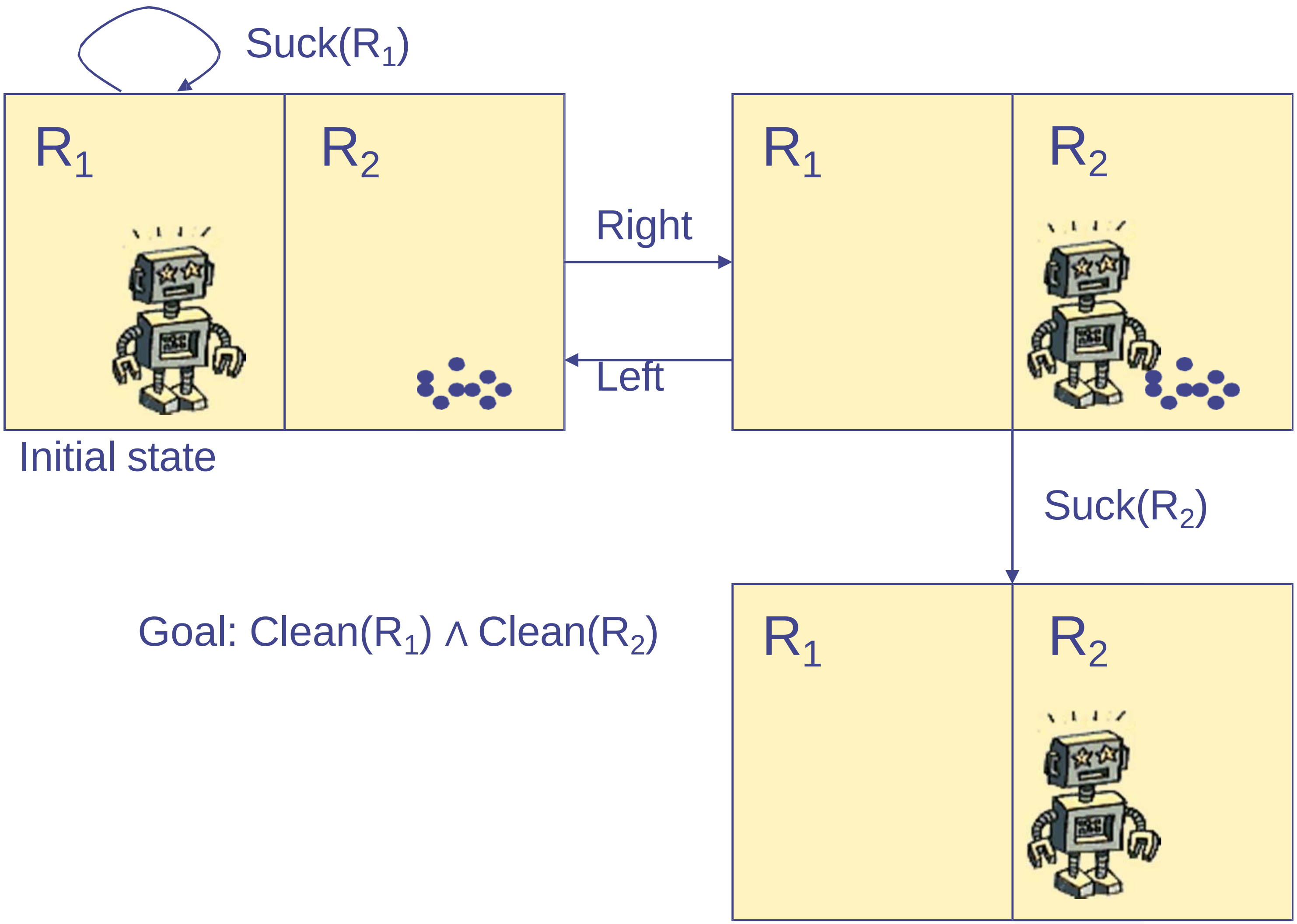
D = $Holding(Key), In(Key,R_1)$

A = $In(Key,Box)$

Part II

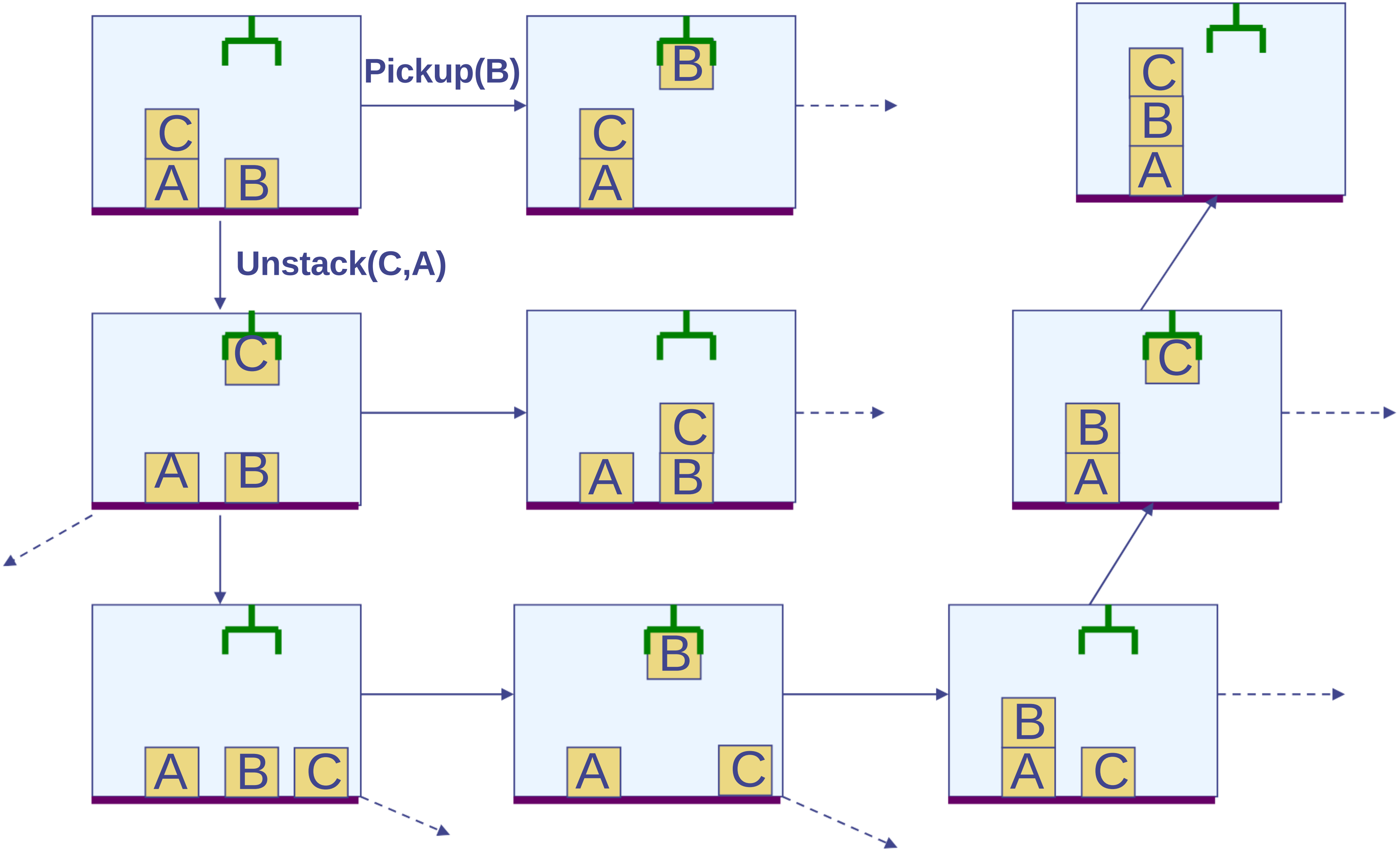
PLANNING METHODS

Forward Planning



Forward Planning

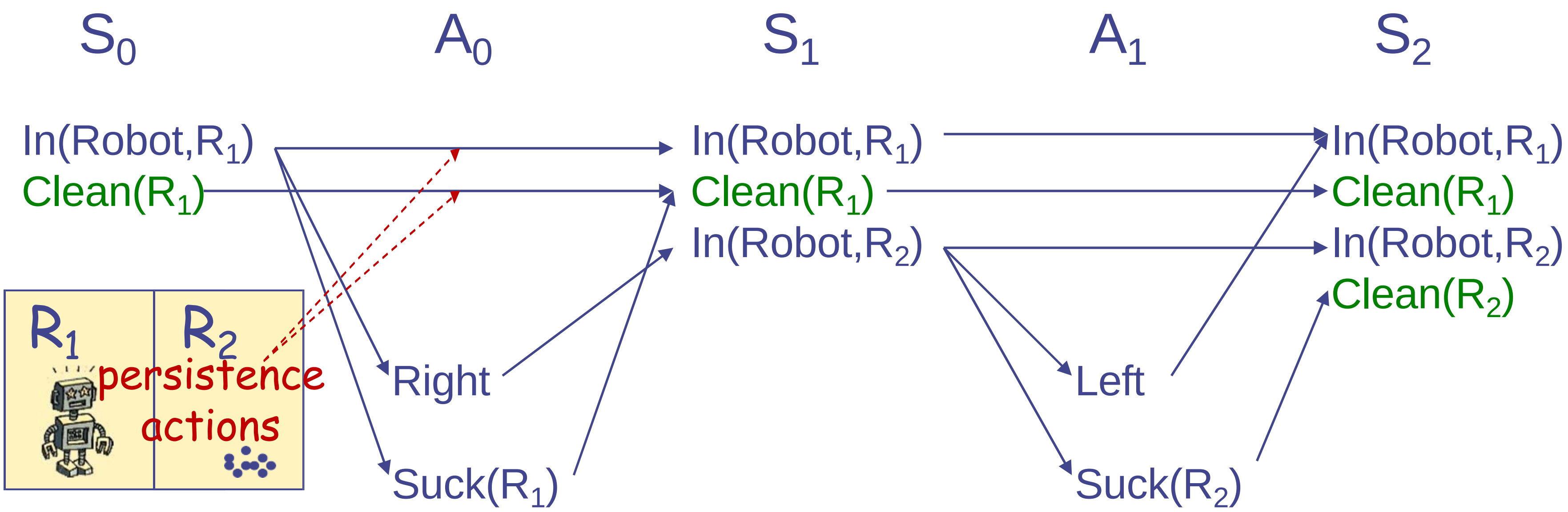
Goal: On(B,A) ∧ On(C,B)



Need for an Accurate Heuristic

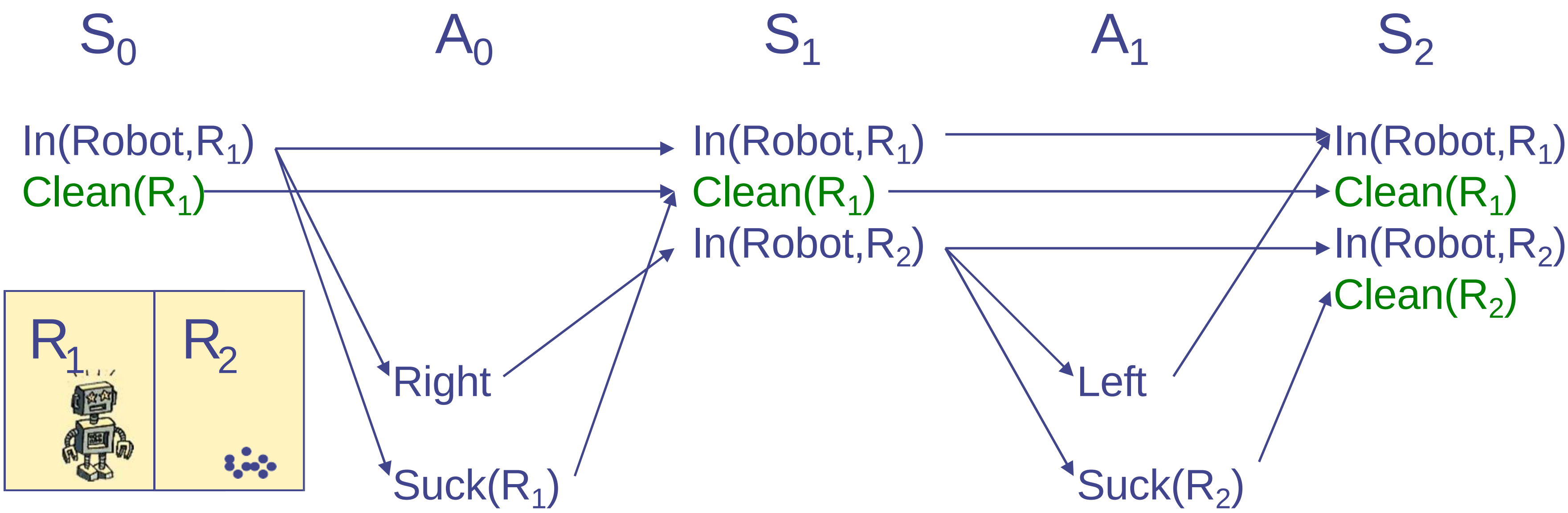
- Forward planning simply searches the space of world states from the initial to the goal state
- Imagine an agent with a large library of actions, whose goal is G, e.g., G = *Have(Milk)*
- In general, many actions are applicable to any given state, so the branching factor is huge
- In any given state, most applicable actions are irrelevant to reaching the goal *Have(Milk)*
- Fortunately, an accurate consistent heuristic can be computed using planning graphs

Planning Graph for a State of the Vacuum Robot



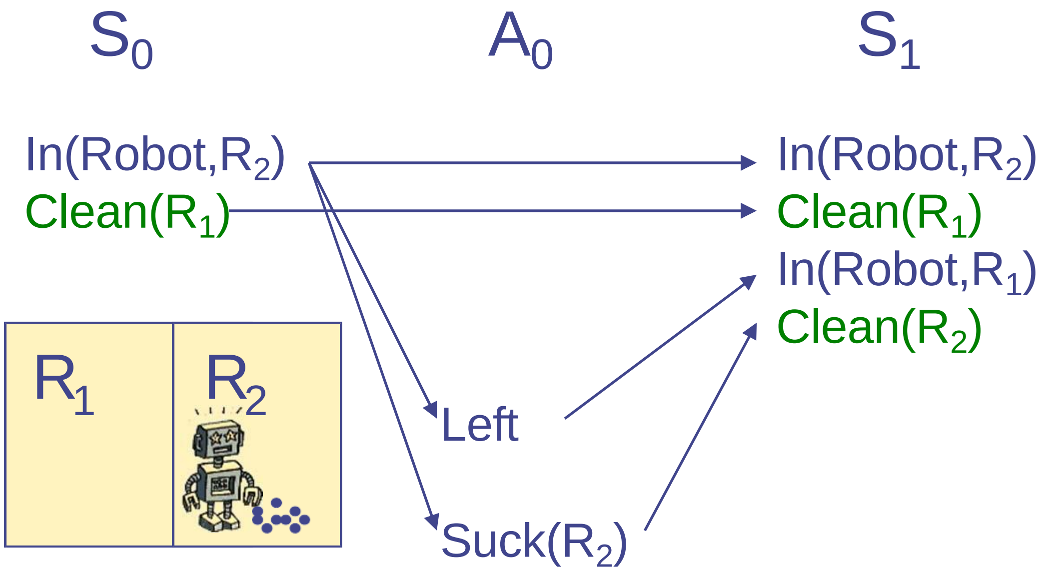
- S_0 contains the state's propositions (here, the initial state)
- A_0 contains all actions whose preconditions appear in S_0
- S_1 contains all propositions that were in S_0 or are contained in the add lists of the actions in A_0
- So, S_1 contains all propositions that may be true in the state reached after the first action
- A_1 contains all actions not already in A_0 whose preconditions appear in S_1, hence that may be executable in the state reached after executing the first action. Etc...

Planning Graph for a State of the Vacuum Robot



- The value of i such that S_i contains all the goal propositions is called the level cost of the goal (here i=2)
- By construction of the planning graph, it is a lower bound on the number of actions needed to reach the goal
- In this case, 2 is the actual length of the shortest path to the goal

Planning Graph for a State of the Vacuum Robot



- The level cost of the goal is 1, which again is the actual length of the shortest path to the goal

Application of Planning Graphs to Forward Planning

- Whenever a new node is generated, compute the planning graph of its state

  Update the planning graph at the parent node

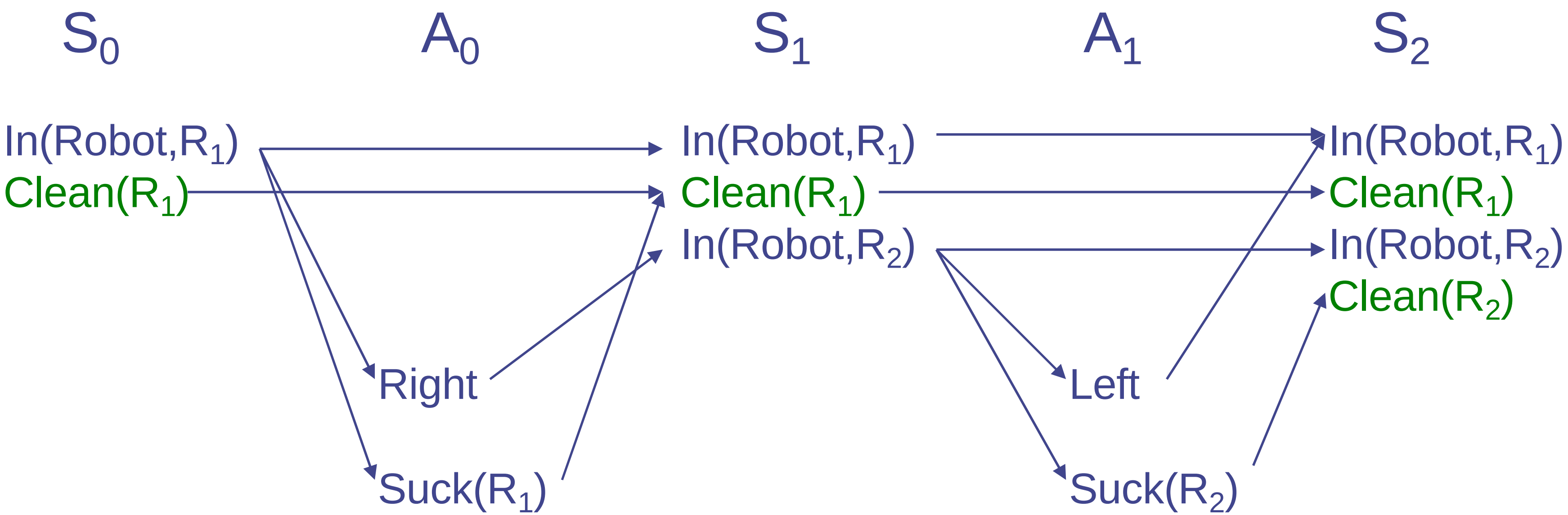
- Stop computing the planning graph when:
  - Either the goal propositions are in a set S_i

    Then i is the level cost of the goal
  - Or when $S_{i+1} = S_i$

    Then the generated node is **not** on a solution path

- Set the heuristic $h(N)$ of a node *N* to the level cost of the goal for the state of *N*
- *h* is a consistent heuristic for unit-cost actions
- Hence, *A** using *h* yields a solution with minimum number of actions

Size of Planning Graph



- An action appears at most once
- A proposition is added at most once and each S_k $(k \neq i)$ is a strict superset of S_{k-1}
- So, the number of levels is bounded by
  $$Min\{number\ of\ actions, number\ of\ propositions\}$$
- In contrast, the state space can be exponential in the number of propositions
- The computation of the planning graph may save a lot of unnecessary search work

Improvement of Planning Graph:

Mutual Exclusions

- Goal: Refine the level cost of the goal to be a more accurate estimate of the number of actions needed to reach it
- Method: Detect obvious exclusions among propositions at the same level (see R&N)
- It usually leads to more accurate heuristics, but the planning graphs can be bigger and more expensive to compute

- Forward planning still suffers from an excessive branching factor
- In general, there are much fewer actions that are relevant to achieving a goal than actions that are applicable to a state
- How to determine which actions are relevant? How to use them?

→ Backward planning

Goal-Relevant Action

- An action is relevant to achieving a goal if a proposition in its add-list matches a sub-goal proposition
- For example:

Stack(B,A)

P = Holding(B) $\wedge$ Block(B) $\wedge$ Block(A) $\wedge$ Clear(A)

D = Clear(A), Holding(B),

A = On(B,A), Clear(B), Handempty

is relevant to achieving On(B,A) $\wedge$ On(C,B)

Regression of a Goal

The regression of a goal G through an action A is the least constraining precondition $R[G,A]$ such that:

If a state S satisfies $R[G,A]$ then:

1. The precondition of A is satisfied in S
2. Applying A to S yields a state that satisfies G

Example

- G = On(B,A) ∧ On(C,B)
- **Stack(C,B)**

  P = Holding(C) ∧ Block(C) ∧ Block(B) ∧ Clear(B)

  D = Clear(B), Holding(C)

  A = On(C,B), Clear(C), Handempty
- R[G,Stack(C,B)] =

  On(B,A) ∧

  Holding(C) ∧ Block(C) ∧ Block(B) ∧ Clear(B)

Example

- G = On(B,A) $\wedge$ On(C,B)
- **Stack(C,B)**

  P = Holding(C) $\wedge$ Block(C) $\wedge$ Block(B) $\wedge$ Clear(B)

  D = Clear(B), Holding(C)

  A = On(C,B), Clear(C), Handempty
- R[G,Stack(C,B)] =

  On(B,A) $\wedge$

  Holding(C) $\wedge$ Block(C) $\wedge$ Block(B) $\wedge$ Clear(B)

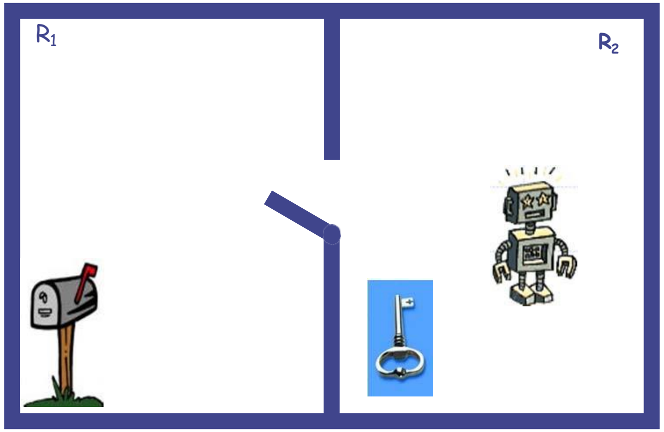
Another Example

- G = In(key,Box) ∧ Holding(Key)
- **Put-Key-Into-Box**

  P = In(Robot,R_1) ∧ Holding(Key)

  D = Holding(Key), In(Key,R_1)

  A = In(Key,Box)
- R[G,Put-Key-Into-Box] = **??**



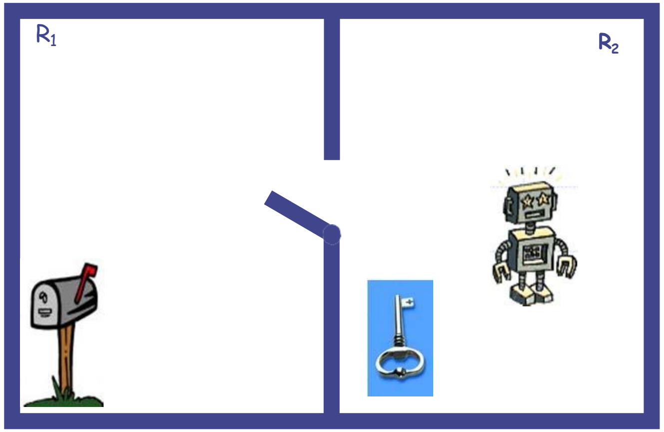
Another Example

- G = In(key,Box) ∧ Holding(Key)
- **Put-Key-Into-Box**

  P = In(Robot,R_1) ∧ Holding(Key)

  D = Holding(Key), In(Key,R_1)

  A = In(Key,Box)



- R[G,Put-Key-Into-Box] = False

  where False is the un-achievable goal

- This means that In(key,Box) ∧ Holding(Key) can't be achieved by executing **Put-Key-Into-Box**

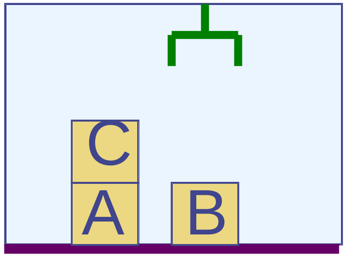
Computation of R[G,A]

1. If any sub-goal of *G* is in *A*’s delete-list then return *False*
2. Else
   a. *G*' ⇐ Precondition of *A*
   b. For every sub-goal *SG* of *G* do:

      If *SG* is not in *A*’s add-list
      then add *SG* to *G'*
3. Return *G'*

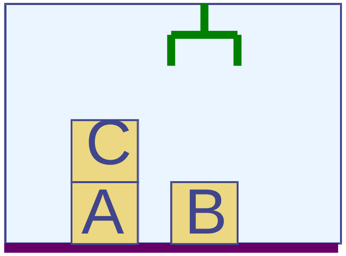
Backward Planning

On(B,A) ∧ On(C,B)



Initial state

Backward Planning



Initial state

On(B,A) ∧ On(C,B)

Stack(C,B)

On(B,A) ∧ Holding(C) ∧ Clear(B)

Pickup(C)

On(B,A) ∧ Clear(B) ∧ Handempty ∧ Clear(C) ∧ On(C,Table)

Stack(B,A)

Clear(C) ∧ On(C,TABLE) ∧ Holding(B) ∧ Clear(A)

Pickup(B)

Clear(C) ∧ On(C,Table) ∧ Clear(A) ∧ Handempty ∧ Clear(B) ∧ On(B,Table)

Putdown(C)

Clear(A) ∧ Clear(B) ∧ On(B,Table) ∧ Holding(C)

Unstack(C,A)

Clear(B) ∧ On(B,Table) ∧ Clear(C) ∧ Handempty ∧ On(C,A)

Backward Planning

On(B,A) ∧ On(C,B)

Stack(C,B)

On(B,A) ∧ Holding(C) ∧ Clear(B)

Pickup(C)

On(B,A) ∧ Clear(B) ∧ Handempty ∧ Clear(C) ∧ On(C,Table)

Stack(B,A)

Clear(C) ∧ On(C,TABLE) ∧ Holding(B) ∧ Clear(A)

Pickup(B)

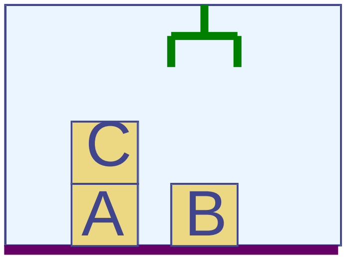
Clear(C) ∧ On(C,Table) ∧ Clear(A) ∧ Handempty ∧ Clear(B) ∧ On(B,Table)

Putdown(C)

Clear(A) ∧ Clear(B) ∧ On(B,Table) ∧ Holding(C)

Unstack(C,A)

Clear(B) ∧ On(B,Table) ∧ Clear(C) ∧ Handempty ∧ On(C,A)



Initial state

Search Tree

- Backward planning searches a space of goals from the original goal of the problem to a goal that is satisfied in the initial state
- There are often much fewer actions relevant to a goal than there are actions applicable to a state
  - ➡ smaller branching factor than in forward planning
- The lengths of the solution paths are the same

Consistent Heuristic for Backward Planning

- A consistent heuristic is obtained as follows :
  - Pre-compute the planning graph of the initial state until it levels off
  - For each node *N* added to the search tree, set *h(N)* to the level cost of the goal associated with *N*
- If the goal associated with *N* can't be satisfied in any set S_k of the planning graph, it can't be achieved, so prune it!
- A single planning graph is computed

How Does Backward Planning Detect Dead-Ends?

$$On(B,A) \land On(C,B)$$

$\downarrow$ **Stack(C,B)**

How Does Backward Planning Detect Dead-Ends?

$$On(B,A) \wedge On(C,B)$$

$$\downarrow \textbf{Stack(B,A)}$$

$$Holding(B) \wedge Clear(A) \wedge On(C,B)$$

$$\downarrow \textbf{Stack(C,B)}$$

$$Holding(B) \wedge Clear(A) \wedge Holding(C) \wedge Clear(B)$$

$$\downarrow \textbf{Pick(B)} \quad \text{[delete list contains Clear(B)]}$$

False

How Does Backward Planning Detect Dead-Ends?

$$On(B,A) \wedge On(C,B)$$

$$\Big\downarrow \text{ Stack(B,A)}$$

$$\boxed{Holding(B)} \wedge Clear(A) \wedge \boxed{On(C,B)}$$

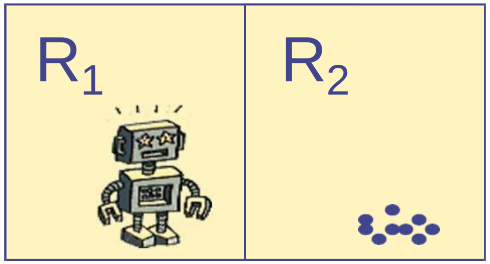
A state constraint such as $Holding(x) \rightarrow \neg(\exists y)On(y,x)$ would have made it possible to prune the path earlier

Part III

SOME EXTENSIONS OF STRIPS LANGUAGE

Extensions of STRIPS

1. Negated propositions in a state



$In(Robot, R_1) \wedge \neg In(Robot, R_2) \wedge Clean(R_1) \wedge \neg Clean(R_2)$

Dump-Dirt(r)

$P = In(Robot, r) \wedge Clean(r)$

$E = \neg Clean(r)$

Suck(r)

$P = In(Robot, r) \wedge \neg Clean(r)$

$E = Clean(r)$

- Q in E means delete ¬Q and add Q to the state
- ¬Q in E means delete Q and add ¬Q

Open world assumption: A proposition in a state is true if it appears positively and false otherwise. A non-present proposition is unknown

Planning methods can be extended rather easily to handle negated proposition but state descriptions are often much longer (e.g., imagine if there were 10 rooms in the above example)

2. Equality/Inequality Predicates

Blocks world:

Move(x,y,z)

P = Block(x) ∧ Block(y) ∧ Block(z) ∧ On(x,y) ∧ Clear(x)
∧ Clear(z) ∧ (x≠z)

D = On(x,y), Clear(z)

A = On(x,z), Clear(y)

Move(x,Table,z)

P = Block(x) ∧ Block(z) ∧ On(x,Table) ∧ Clear(x)
∧ Clear(z) ∧ (x≠z)

D = On(x,y), Clear(z)

A = On(x,z)

Move(x,y,Table)

P = Block(x) ∧ Block(y) ∧ On(x,y) ∧ Clear(x)

D = On(x,y)

A = On(x,Table), Clear(y)

2. Equality/Inequality Predicates

Blocks world:

Move(x,y,z)

P = Block(x) ∧ Block(y) ∧ Block(z) ∧ On(x,y) ∧ Clear(x)
∧ Clear(z) ∧ (x≠z)

D = On(x,y), Clear(z)

A = On(x,z), Clear(y)

Move(x,Table,z)

P = Block(x) ∧ Block(z) ∧
∧ Clear(z) ∧ (x≠z)

D = On(x,y), Clear(z)

A = On(x,z)

Move(x,y,Table)

P = Block(x) ∧ Block(y) ∧

D = On(x,y)

A = On(x,Table), Clear(y)

Planning methods simply evaluate (x≠z) when the two variables are instantiated

This is equivalent to considering that propositions (A ≠B) , (A ≠C) , are implicitly true in any state

3. Algebraic expressions

Two flasks F_1 and F_2 have volume capacities of 30 and 50, respectively

F_1 contains volume 20 of some liquid

F_2 contains volume 15 of this liquid

State:

$Cap(F_1,30) \wedge Cont(F_1,20) \wedge Cap(F_2, 50) \wedge Cont(F_2,15)$

Action of pouring a flask into the other:

Pour(f,f')

$P = Cont(f,x) \wedge Cap(f',c') \wedge Cont(f',y) \quad \wedge \quad (f \neq f')$

$D = Cont(f,x), Cont(f',y),$

$A = Cont(f,\max\{x+y-c',0\}), Cont(f',\min\{x+y,c'\})$

3. Algebraic expressions

Two flasks F_1 and F_2 have volume capacities of 30 and 50, respectively

F_1 contains volume 20 of some liquid

F_2 contains volume 15 of this liquid

This extension requires extension methods equipped with algebraic manipulation capabilities

State:

Cap(F_1, … t(F_2,15)

Action of pouring a flask into the other:

Pour(f,f')

P = Cont(f,x) ∧ Cap(f',c') ∧ Cont(f',y) ∧ (f ≠ f')

D = Cont(f,x), Cont(f',y),

A = Cont(f,max{x+y-c',0}), Cont(f',min{x+y,c'})

Extensions of STRIPS

4. State Constraints

h	b	
c	d	g
e	a	f

State:

Adj(1,2) ∧ Adj(2,1) ∧ ... ∧ Adj(8,9) ∧ Adj(9,8) ∧
At(h,1) ∧ At(b,2) ∧ At(c,4) ∧ ... ∧ At(f,9) ∧ Empty(3)

Move(x,y,z)

P = At(x,y) ∧ Empty(z) ∧ Adj(y,z)
D = At(x,y), Empty(z)
A = At(x,z), Empty(y)

Extensions of STRIPS

4. State Constraints

h	b	
c	d	g
e	a	f

State:

Adj(1,2) ∧ ~~Adj(2,1) ∧~~ ... ∧ Adj(8,9) ∧ ~~Adj(9,8) ∧~~

At(h,1) ∧ At(b,2) ∧ At(c,4) ∧ ... ∧ At(f,9) ∧ Empty(3)

State constraint:

Adj(x,y) → Adj(y,x)

Move(x,y,z)

P = At(x,y) ∧ Empty(z) ∧ Adj(y,z)

D = At(x,y), Empty(z)

A = At(x,z), Empty(y)

More Complex State Constraints in 1st-Order Predicate Logic

Blocks world:

$(\forall x)[Block(x) \quad \wedge \neg(\exists y)On(y,x) \wedge \neg Holding(x)] \quad \rightarrow Clear(x)$

$(\forall x)[Block(x) \quad \wedge Clear(x)] \rightarrow \neg(\exists y)On(y,x) \wedge \neg Holding(x)$

$Handempty \leftrightarrow \neg(\exists x)Holding(x)$

would simplify greatly the description of the actions

State constraints require planning methods with logical deduction capabilities, to determine whether goals are achieved or preconditions are satisfied

Some Applications of AI Planning

- Military operations
- Operations in container ports
- Construction tasks
- Machining and manufacturing
- Autonomous control of satellites and other spacecrafts