, 2022

Planning

Classical Planning

Bilal Hoteit

Table of contents

Introduction.

- Planning :

How to select and organize a sequence of actions to achieve a Goal.

Planning is a topic of traditional interest in Artificial Intelligence as it is an important part of many different AI applications, such as robotics and intelligent agents.

To be able to plan, a system needs to be able to reason about the individual and cumulative effects of a series of actions. This is a skill that is only observed in a few animal species and only mastered by humans.

The planning problems we will be discussing today are mostly Toy-World problems but they can be scaled up to real-world problems such as a robot negotiating a space.

Introduction.

- Planning :

Classical VS Dynamic.

Environment classification play an important part on the techniques and tools used to support planning.

- Planning :

Deterministic – Temporal – Contingent – Probabilistic - Multiple Agents.

Each one has its own of techniques and tools used to provide impractical solution.

- Planning : Planning research has been central to AI.

Papers on planning are a staple of mainstream AI journals and conferences.

Introduction.

- Planning :

  How to select and organize a sequence of actions to achieve a Goal.

- Planning :

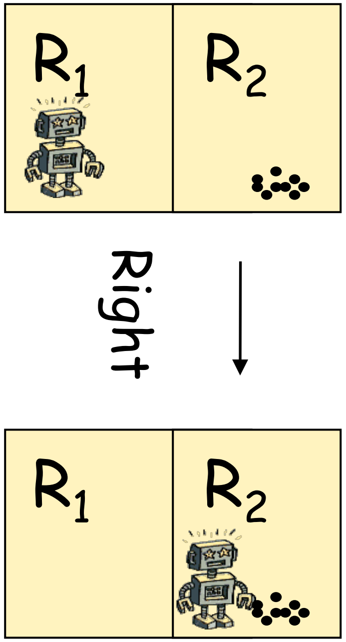
  State

  Successor function

  Goal test

  Solution

  Heuristic function



Introduction.

Standard Search vs. Specific Planning systems

- Planning :
  - State
  - Successor function
  - Goal test
  - Solution
  - Heuristic function

- Planning :
  - State-Space Search.
  - Plan-Space Search.

1- Problem-Solving Search.

2- CSP Search.

3- Planning Search

Introduction.

- Planning :

How to select and organize a sequence of actions to achieve a Goal.

- Planning :

State : A possible world (**full assignments** to a set of variables/ features).

S(A, B, C) . Assignment..... S(F, F, F)

Goal: Agent wants to be in a possible world were some variables are given specific values

S(T, T, T)

Introduction.

- Planning :

How to select and organize a sequence of actions to achieve a Goal.

- Planning :

Action : take the agent from one state to another

S(F, F, F) $\xrightarrow{\text{A1}}$ S(T, F, F) $\xrightarrow{\text{A2}}$ S(T, T, F)

Solution: sequence of actions that when performed will take the agent from the current state to a goal state

S(F, F, F) A1 A2 A3 S(T, T, T)

Introduction.

- 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

Knowledge Represenations.

- Classical representation (Atomic style, A set of propositions).

- STRIPS representation and assumption (STanford Research Institute Problem Solver). (Factored style, At most a FOL).

- State-Variable representation. (Factored style, A tuple os state variables, More expressive language)

    - ADL representation (Action Description Language).
    - PDDL representation (Planning Domain Definition Language)

Planning using STRIPS.

Planning using STRIPS.

The “classical” approach most planners use today is derived from the STRIPS language.

STRIPS was devised by SRI in the early 1970s to control a robot called Shakey.

Shakey’s task was to negotiate a series of rooms, move boxes, and grab objects.

The STRIPS language was used to derive plans that would control Shakey’s movements so that he could achieve his goals.

The STRIPS language is very simple but expressive language that lends itself to efficient planning algorithms.

The representation used in Prolog is derived from the original STRIPS representation.

STRIPS Language through Examples.

When beginning to produce a planner there are certain representation considerations that need to be made:

How do we represent the state of the world?

How do we represent operators?

Does our representation make it easy to:

- Check preconditions;
- Alter the state of the world after performing actions;
- Recognize the goal state?

Propositional logic VS FOL.

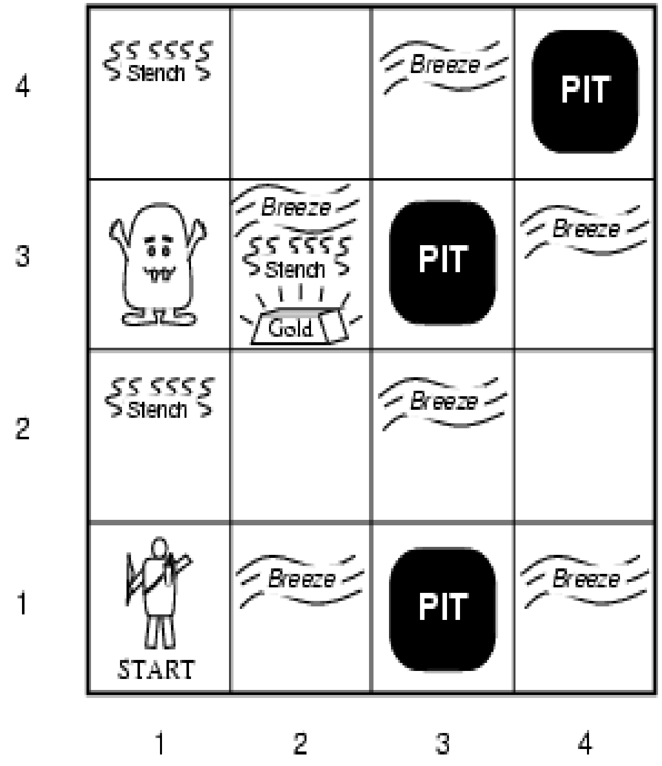
Propositional logic lacks the expressive power to concisely describe an environment with many objects.

AS writing a separate rule about breezes and pits for each square. B1,1 ⇔ (P1,2 ∨ P2,1) .

The atomic sentence consist of a single proposition symbol that can be True or False. Remember that symbols such as W1,3 are atomic.

Complex sentences are constructed from atomic sentences, using parentheses and logical connectives.

There are five connectives in common use: ¬, ∧, ∨, ⇒, ⇔ .



Remember that LITERAL atomic sentence (a positive symbol) or a negated atomic sentence (a negative symbol).

Propositional logic VS FOL.

FIRST-ORDER LOGIC is sufficiently expressive to represent a good deal of our commonsense knowledge, specifically in a complex environments .

Objects: people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries ...

Relations: these can be unary relations or properties such as red, round, bogus, prime, multistoried ..., or more general n-ary relations such as brother of, bigger than, inside, part of, has color, occurred after, owns, comes between, ...

Functions: father of, best friend, third inning of, one more than, beginning of ….

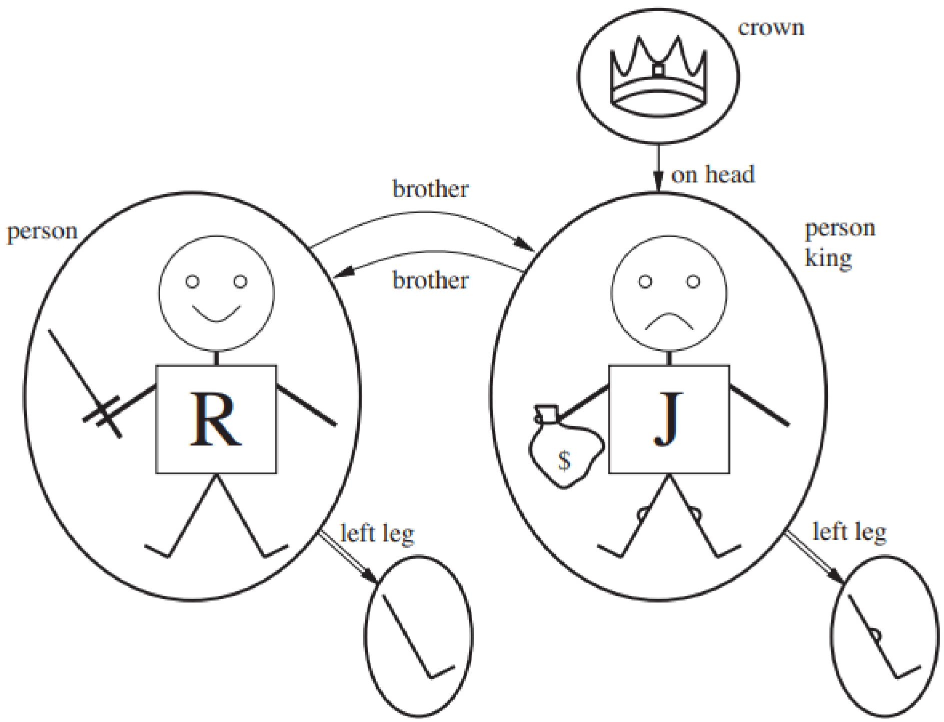
The language of first-order logic, whose syntax and semantics, is built around objects and relations.

The primary difference between propositional and first-order logic lies in the ontological commitment.

Models for FOL.

FIRST-ORDER LOGIC is sufficiently expressive to represent a good deal of our commonsense knowledge, specifically in a complex environments .

The domain of a model is the set of objects or domain elements it contains.



Objects – 5 :
Richard, John, Crown, 2 Leftlegs.

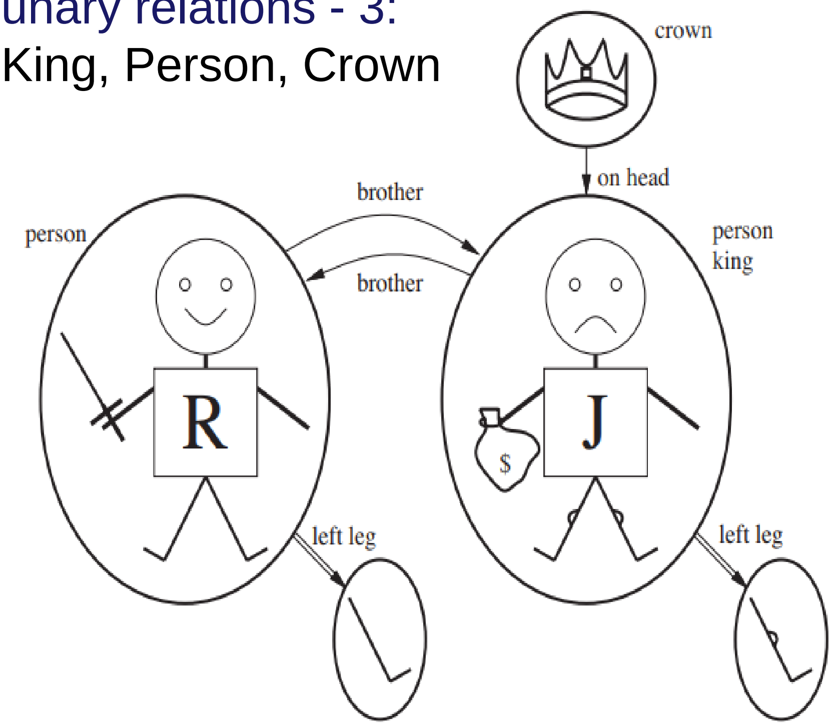
Relations – 5 :
binary relations - 2:
Brother, Onhead,
unary relations - 3:
King, Person, Crown

Functions – 1 :
Unary function: leftleg.

Models for FOL.

Objects – 5 :
Richard, John, Crown, 2 Leftlegs.

Relations – 5 :
binary relations - 2:
Brother, Onhead,
unary relations - 3:
King, Person, Crown



Functions – 1 :
Unary function: leftleg.

Examples:

Person(Richard),

Brother (Richard, John).

LeftLeg(John).

Complex sentences:

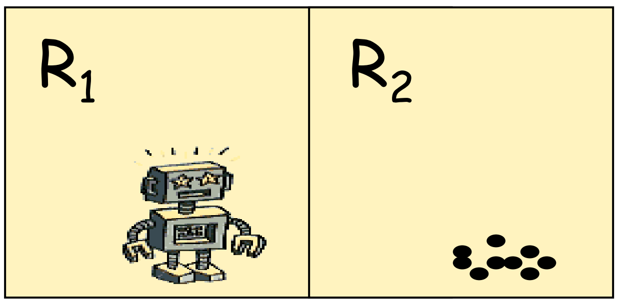
¬Brother (LeftLeg(Richard), John)

Married(Father (Richard), Mother (John))

STRIPS Language through Examples.

STRIPS Language through Examples.

Vacuum-Robot Example



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

Objects:

Room(R1-R2)
Robot(RT)
Dust(D)

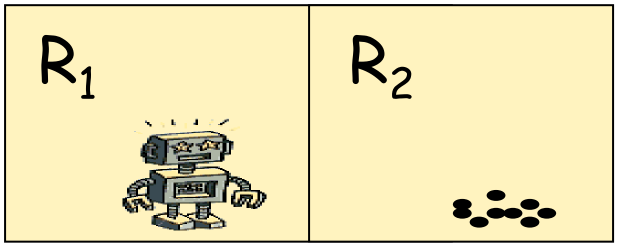
Relations or predicts:

In (RT - Robot, R1 - Room)
Clean(R1- Room)

Functions:

(distance R1 R2 - Room)

STRIPS Language through Examples.



- Two rooms: R_1 and R_2
- A vacuum robot (RT)
- Dust (D)

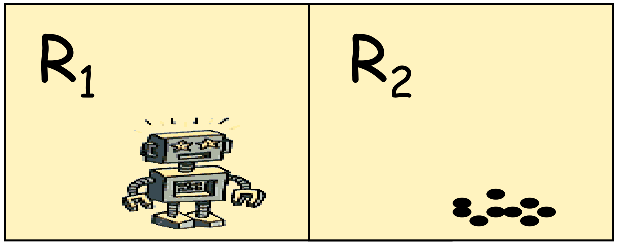
State Representation:

$$\underbrace{In(RT, R_1)}\ \underbrace{\wedge}\ \underbrace{Clean(R_1)}$$

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

Logical "and" connective

STRIPS Language through Examples.

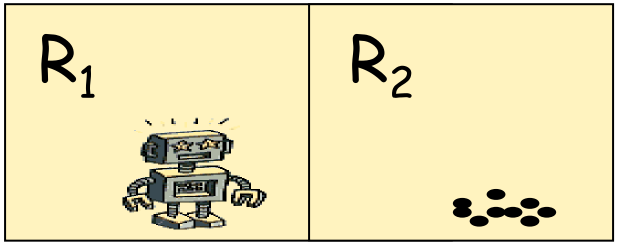


- Two rooms: R_1 and R_2
- A vacuum robot (RT)
- Dust (D)

State Representation: KB

- Conjunction of propositions, such as $In(RT, R_1) \land Clean(R_1)$
- 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 $In(RT,R_1) \lor In(RT,R_2)$
- No variable, e.g., $\exists x\ Clean(x)$

STRIPS Language through Examples.



- Two rooms: R_1 and R_2
- A vacuum robot (RT)
- Dust (D)

Goal Representation

$$Clean(R1) \land Clean(R2)$$

- 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 (called sub-goals) are also in S

STRIPS Language through Examples.

Action Representation

Right

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

Sets of propositions

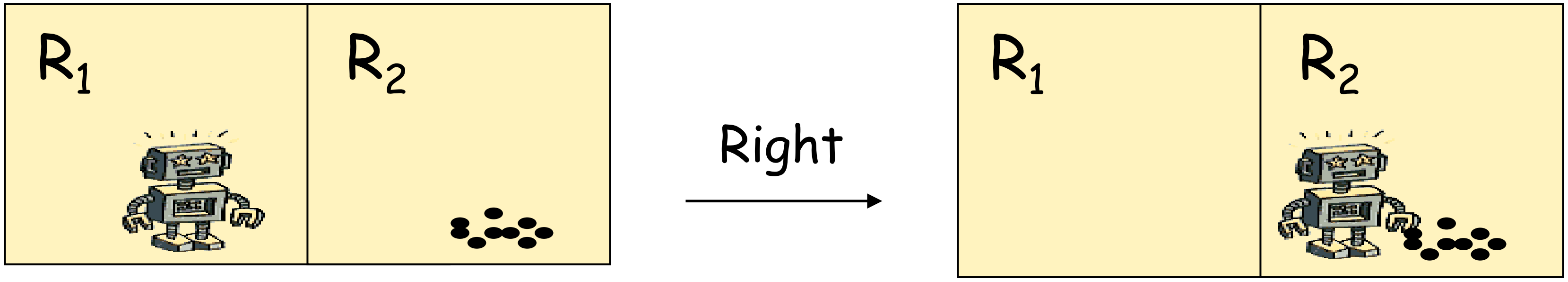
Same form as a goal: conjunction of propositions

STRIPS Language through Examples.

Action Representation

Right

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



$In(RT, R_1) \wedge Clean(R_1)$ $\quad$ $In(RT, R_2) \wedge Clean(R_1)$

STRIPS Language through Examples.

Action Representation

Right

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

- 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

STRIPS Language through Examples.

Action Representation

Left

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

Suck(R_1)

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

Suck(R_2)

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

STRIPS Language through Examples.

Action Representation

Left

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

Suck(r)

- $P = In(RT, r)$
- $D = \varnothing$ [empty list]
- $A = Clean(r)$

STRIPS Language through Examples.

Action Schema

It describes several actions, here: $Suck(R_1)$ and $Suck(R_2)$

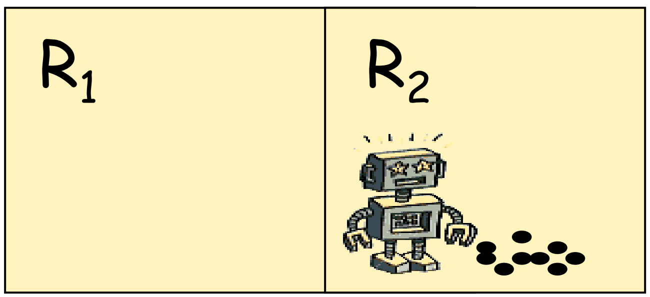
Parameter that will get "instantiated" by matching the precondition against a state

Suck(r)

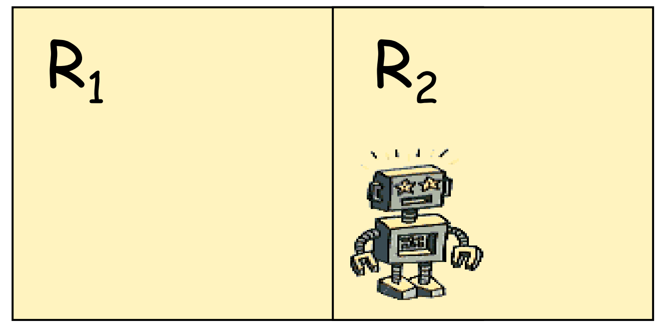
- $P = In(Robot, r)$
- $D = \varnothing$
- $A = Clean(r)$

STRIPS Language through Examples.

Action Schema



$\text{Suck}(R_2) \longrightarrow$



$In(RT, R_2) \wedge Clean(R1)$

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

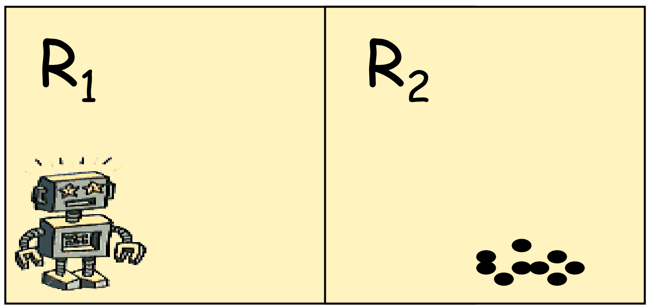
Suck(r)

$r \leftarrow R_2$

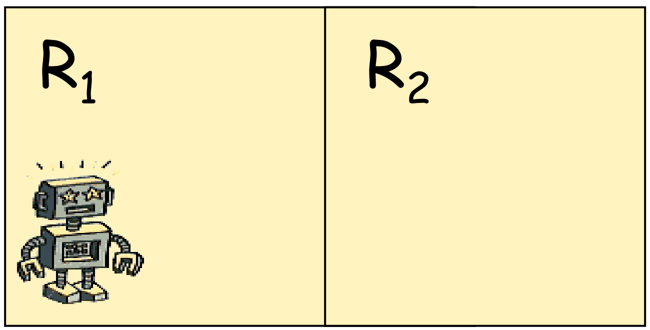
- $P = In(RT, r)$
- $D = \varnothing$
- $A = Clean(r)$

STRIPS Language through Examples.

Action Schema



$\xrightarrow{Suck(R_1)}$



$In(RT, R_1) \wedge Clean(R1)$ $\qquad$ $In(RT, R_1) \wedge Clean(R_1)$

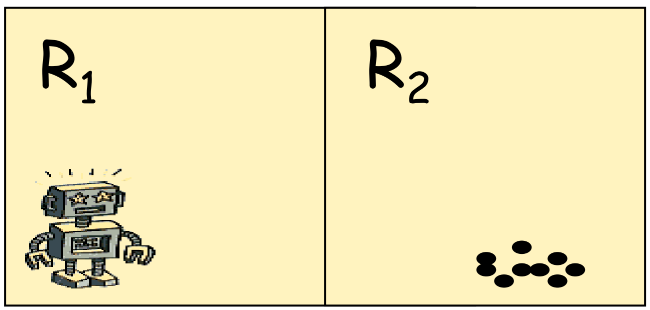
Suck(r)

$r \leftarrow R_1$

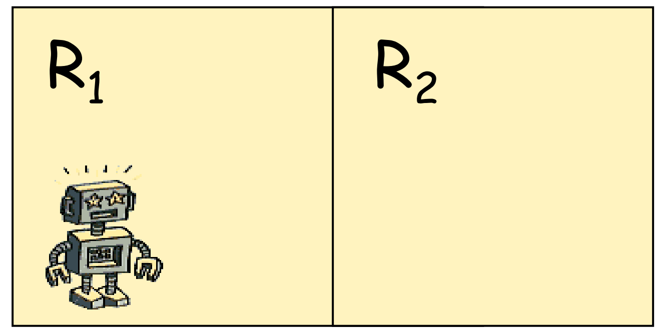
- $P = In(RT, r)$
- $D = \varnothing$
- $A = Clean(r)$

STRIPS Language through Examples.

Action Schema



$\text{Suck}(R_1) \longrightarrow$



$In(RT, R_1) \wedge Clean(R1)$

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

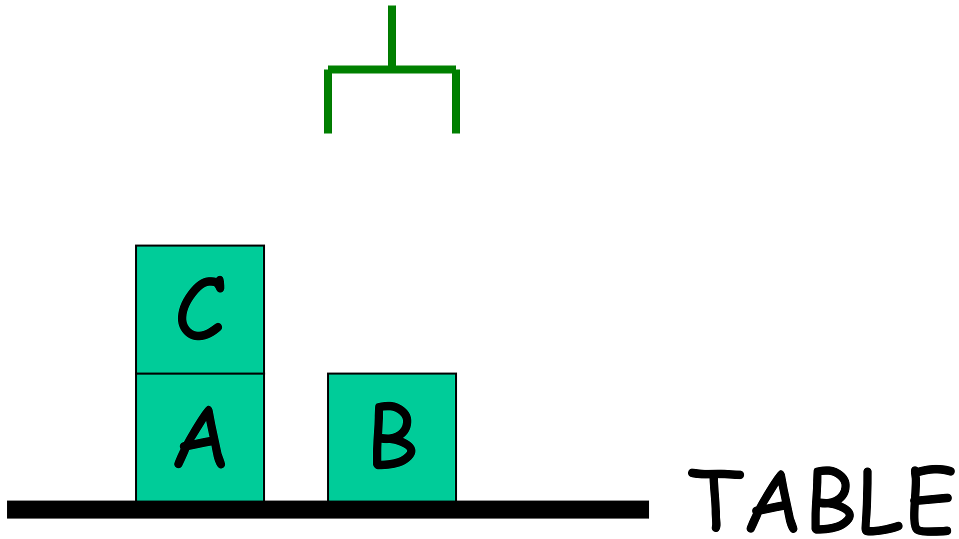
Suck(r)

$r \leftarrow R_1$

- $P = In(RT, r)$
- $D = \varnothing$
- $A = Clean(r)$

STRIPS Language through Examples.

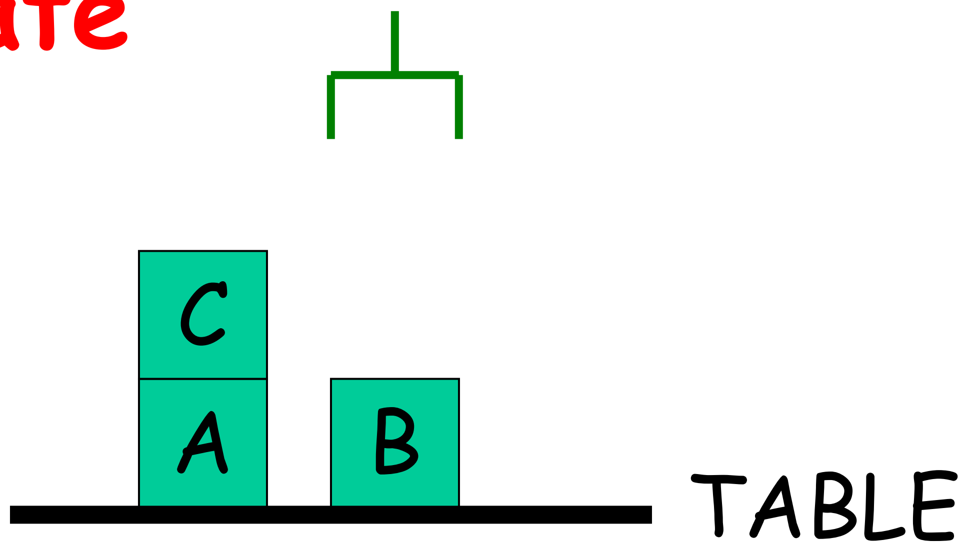
Blocks-World Example



- 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

STRIPS Language through Examples.

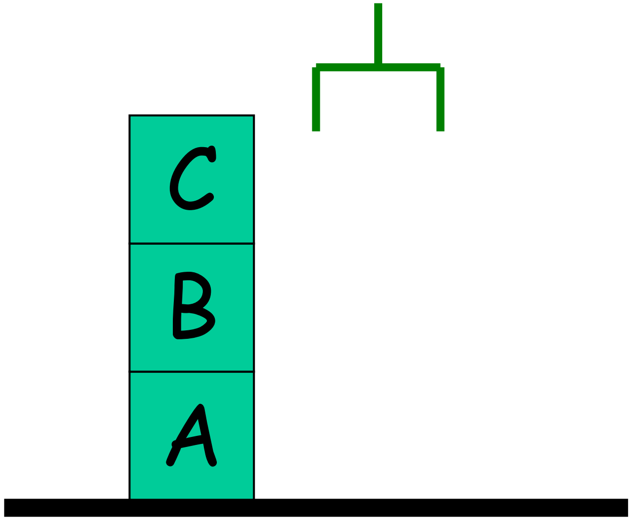
State



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

STRIPS Language through Examples.

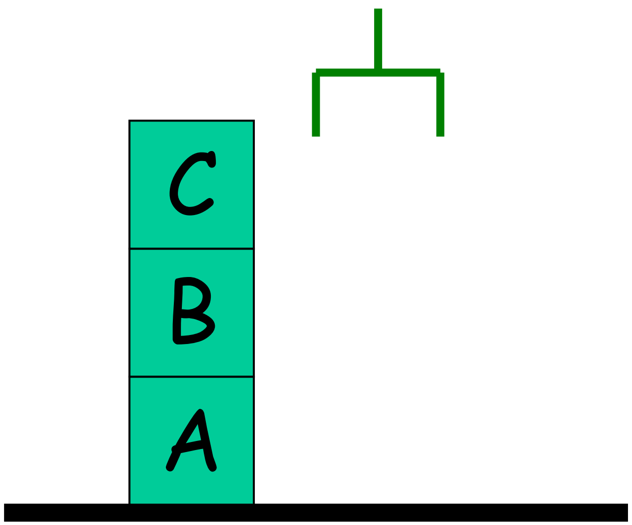
GOAL

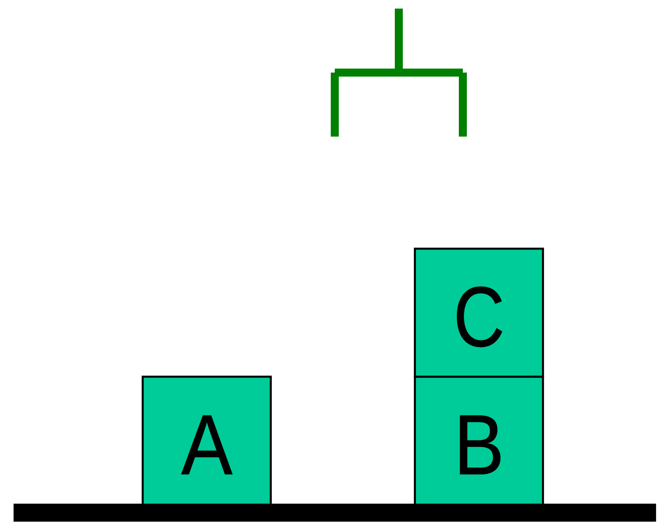


On(A,TABLE) ∧ On(B,A) ∧ On(C,B) ∧ Clear(C)

STRIPS Language through Examples.

GOALs





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

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

STRIPS Language through Examples.

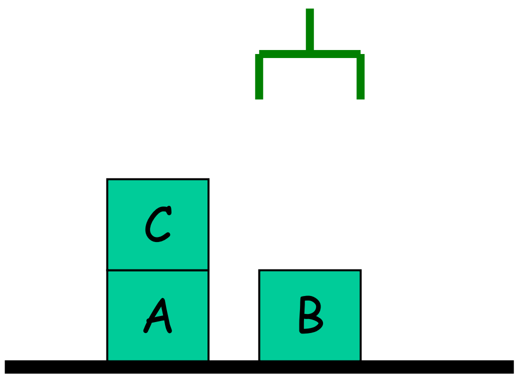
Action

Unstack(x,y)
P = Handempty $\wedge$ Block(x) $\wedge$ Block(y) $\wedge$ Clear(x) $\wedge$ On(x,y)
D = Handempty, Clear(x), On(x,y)
A = Holding(x), Clear(y)

STRIPS Language through Examples.

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)



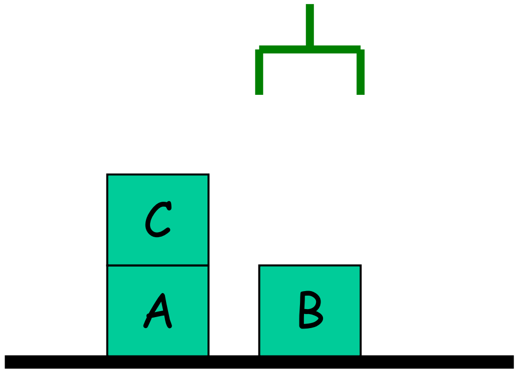
Block(A) ∧ Block(B) ∧ Block(C) ∧ On(A,Table) ∧ On(B,Table) ∧ On(C,A) ∧ Clear(B) ∧ Clear(C) ∧ Handempty

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)

STRIPS Language through Examples.

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)

STRIPS Language through Examples.

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

STRIPS Language through Examples.

All actions

A block can always fit
on the table

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)

Pickup(x)
P = Handempty ∧ Block(x) ∧
Clear(x) ∧ On(x,Table)
D = Handempty, Clear(x), On(x,Table)
A = Holding(x)

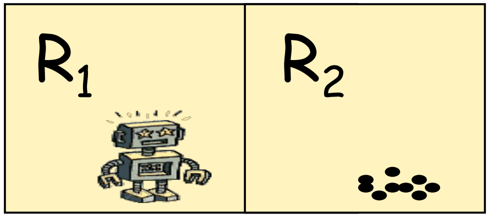
Stack(x,y)
P = Holding(x) ∧ Block(x) ∧ Block(y)
∧ Clear(y)
D = Clear(y), Holding(x)
A = On(x,y), Clear(x), Handempty

Putdown(x)
P = Holding(x), ∧ Block(x)
D = Holding(x)
A = On(x,Table), Clear(x), Handempty

A block can always fit
on the table

Some Extensions of STRIPS.

1. Negated propositions in a state



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

Dump-Dirt(r)

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

$E = \neg Clean(r)$

Suck(r)

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

$E = Clean(r)$

effects means del-list U add-list

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 if it appears negatively. A non-present proposition is unknown

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

Some Extensions of STRIPS.

2. Equality/Inequality Predicates

Blocks world:

Move(x,y,z)

P = Block(x) $\wedge$ Block(y) $\wedge$ Block(z) $\wedge$ On(x,y) $\wedge$ Clear(x) $\wedge$ Clear(z) $\wedge$ (x$\neq$z)

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

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

Move(x,Table,z)

P = Block(x) $\wedge$ Block(z) $\wedge$ On(x,Table) $\wedge$ Clear(x) $\wedge$ Clear(z) $\wedge$ (x$\neq$z)

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

A = On(x,z)

Move(x,y,Table)

P = Block(x) $\wedge$ Block(y) $\wedge$ On(x,y) $\wedge$ Clear(x)

D = On(x,y)

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

Planning methods.

1- Linear search:

a- forward search

b- backward search

2- non-linear search

a- partial order plans

b- hierarchical plans

Forward Search.

To find a plan, a solution: search in the state-space graph.

- The **states** are the **possible worlds**
- The **arcs** from a state ***s*** represent all of the **actions** that are legal in state ***s***.
- A **plan** is a path from the state representing the initial state to a state that satisfies the goal.

What actions **a** are legal/possible in a state **s**?

A. Those where **a**'s effects are satisfied in **s**

B. Those where **a**'s preconditions are satisfied in **s**

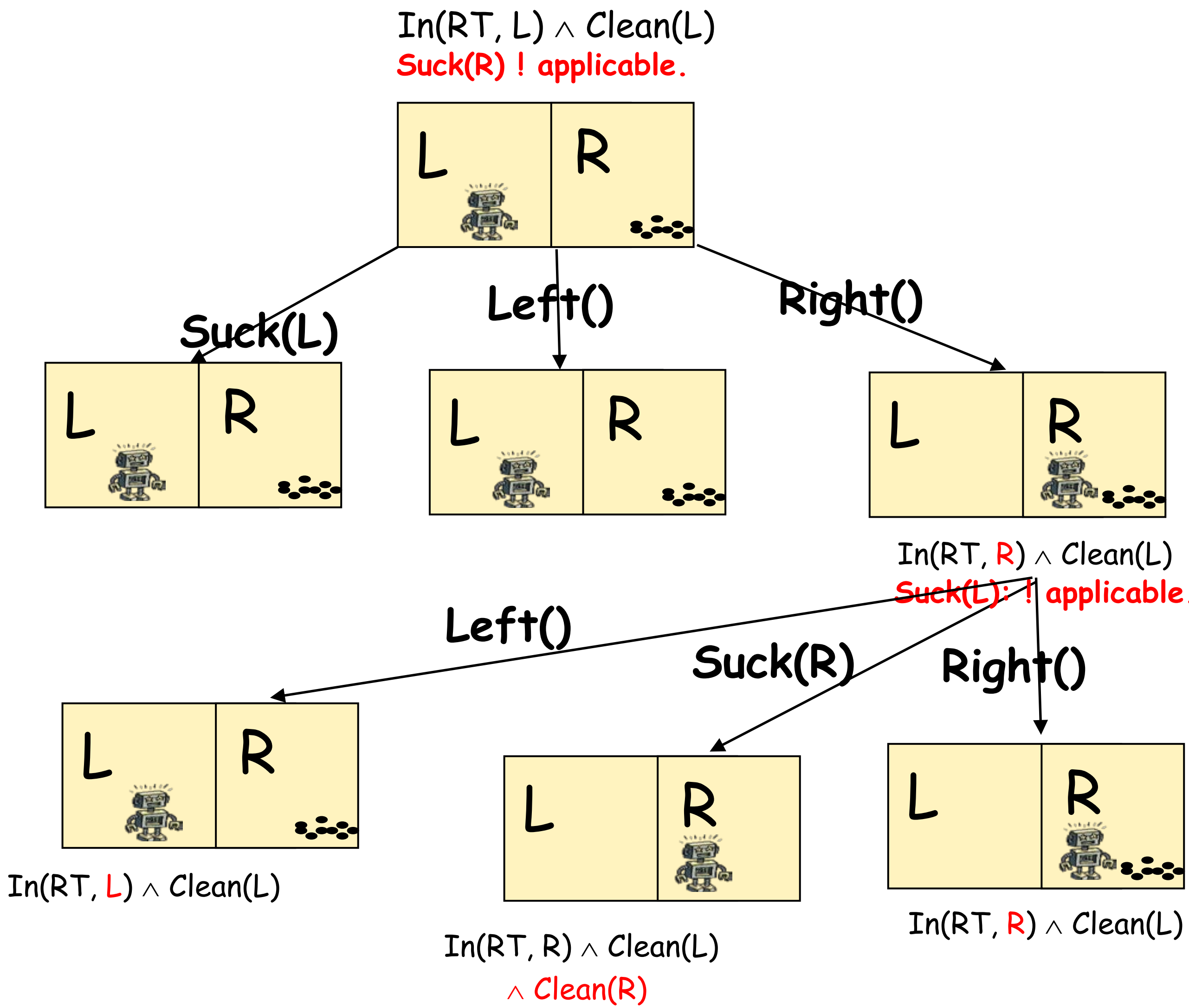
C. Those where the state **s'** reached via **a** is on the way to the goal

Forward Search.

Suck(r)
- P = In(Robot, r)
- E = Clean(r)

Left()
- P = ∅
- E = In(Robot, L)
  ¬ In(Robot, R)

Right()
- P = ∅
- E = In(Robot, R)
- ¬ In(Robot, L)



Forward Search.

1. Look at the state of the world:
    - Is it the goal state? If so, the list of operators so far is the plan to be applied.
    - If not, go to Step 2.

2. Pick an operator:
    - Check that it has not already been applied (to stop looping).
    - Check that the preconditions are satisfied.

    If either of these checks fails, backtrack to get another operator.

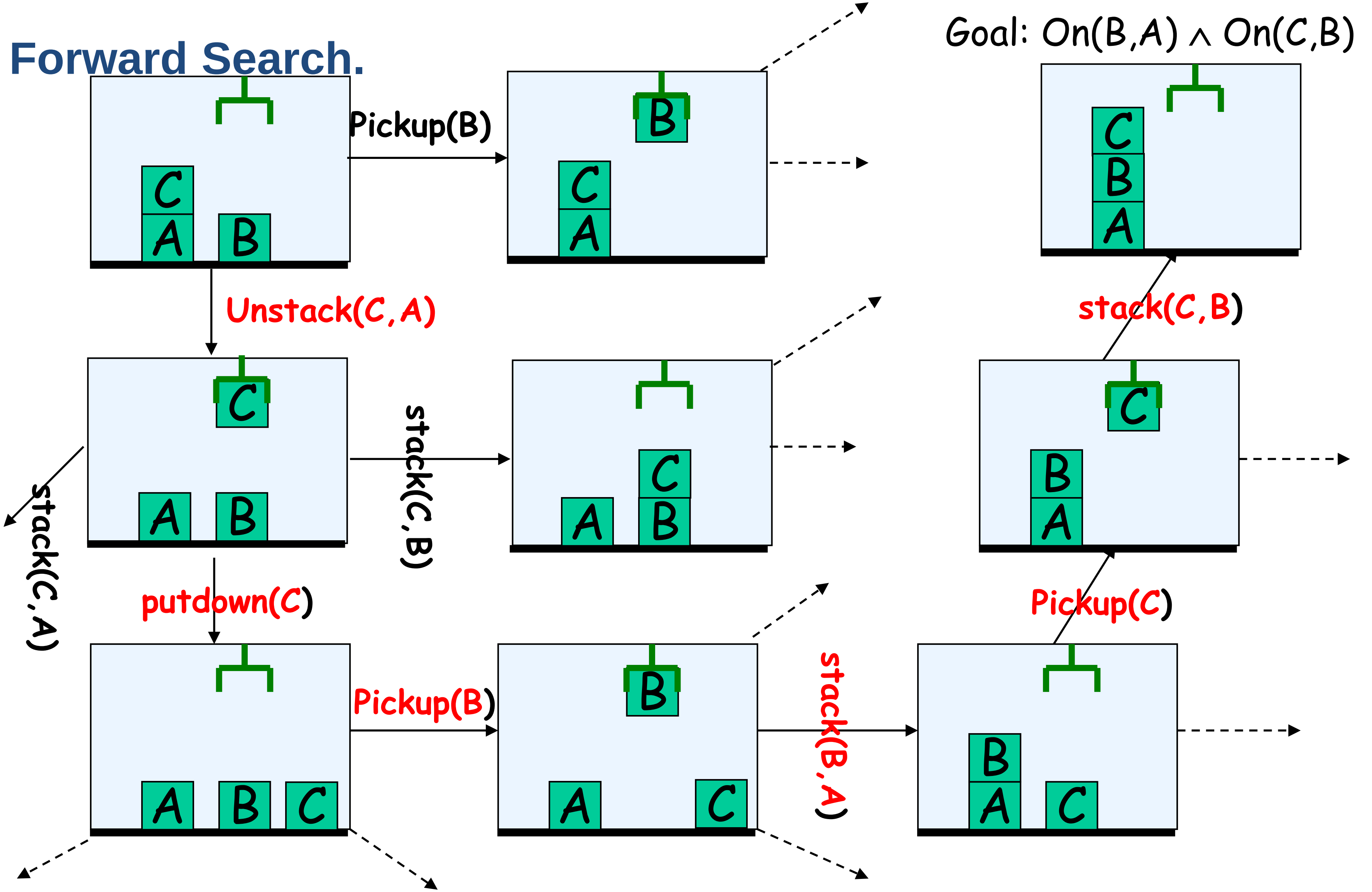
3. Apply the operator:
    1. Make changes to the world: delete from and add to the world state.
    2. Add operator to the list of operators already applied.
    3. Go to Step 1.

Forward Search.

Forward-search (Σ, s0, g)
  s ← s0; π ← hi
  loop
    if s satisfies g, then return π
    A0 ← {a ∈ A | a is applicable in s}
    if A0 = ∅, then return failure
    non-deterministically choose a ∈ A0
    s ← γ(s, a); π ← π.a

A state-variable planning problem is a triple P = (Σ, s0, g).
Σ is a state-variable planning domain.
s0 is a state called the initial state.
g is a set of ground literals called the goal.
A solution for P is any plan π = (a1, . . . , an) such that the state γ(s0, π) satisfies g.

Forward Search.

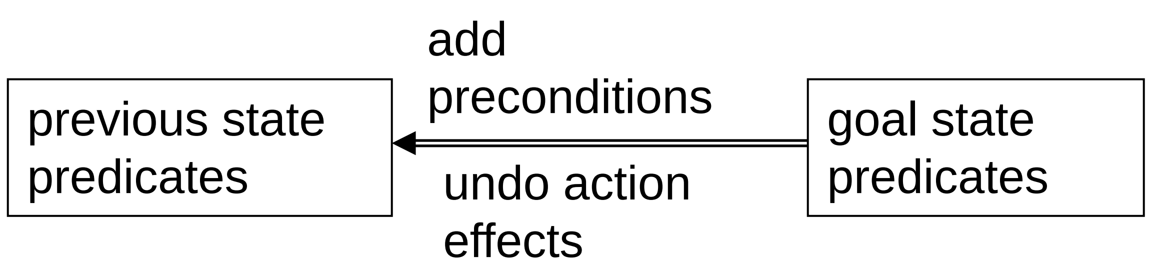


Backward Search.

actions with goal state as ‘effect’ are ‘regressed’

focus on ‘relevant’ actions

avoid actions that undo goal state predicates



Backward Search.

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) ∧ Block(B) ∧ Block(A) ∧ Clear(A)
D = Clear(A), Holding(B),
A = On(B,A), Clear(B), Handempty

is relevant to achieving On(B,A)∧On(C,B)

Backward Search.

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 achieves R[G,A] then:

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

Backward Search.

Regression of a Goal - Example

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

- **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)

Backward Search.

Regression of a Goal - Example

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

- **Stack(C,B)**

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

  $D = Clear(B), Holding(C)$

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

- $R[G,Stack(C,B)] =$

  $On(B,A) \wedge$

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

Backward Search.

Computation of R[G,A]

1. If any sub-goal of G is in A's delete list then return False
2. Else
   a. $G' \leftarrow$ 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 Search.

Computation of R[G,A]

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

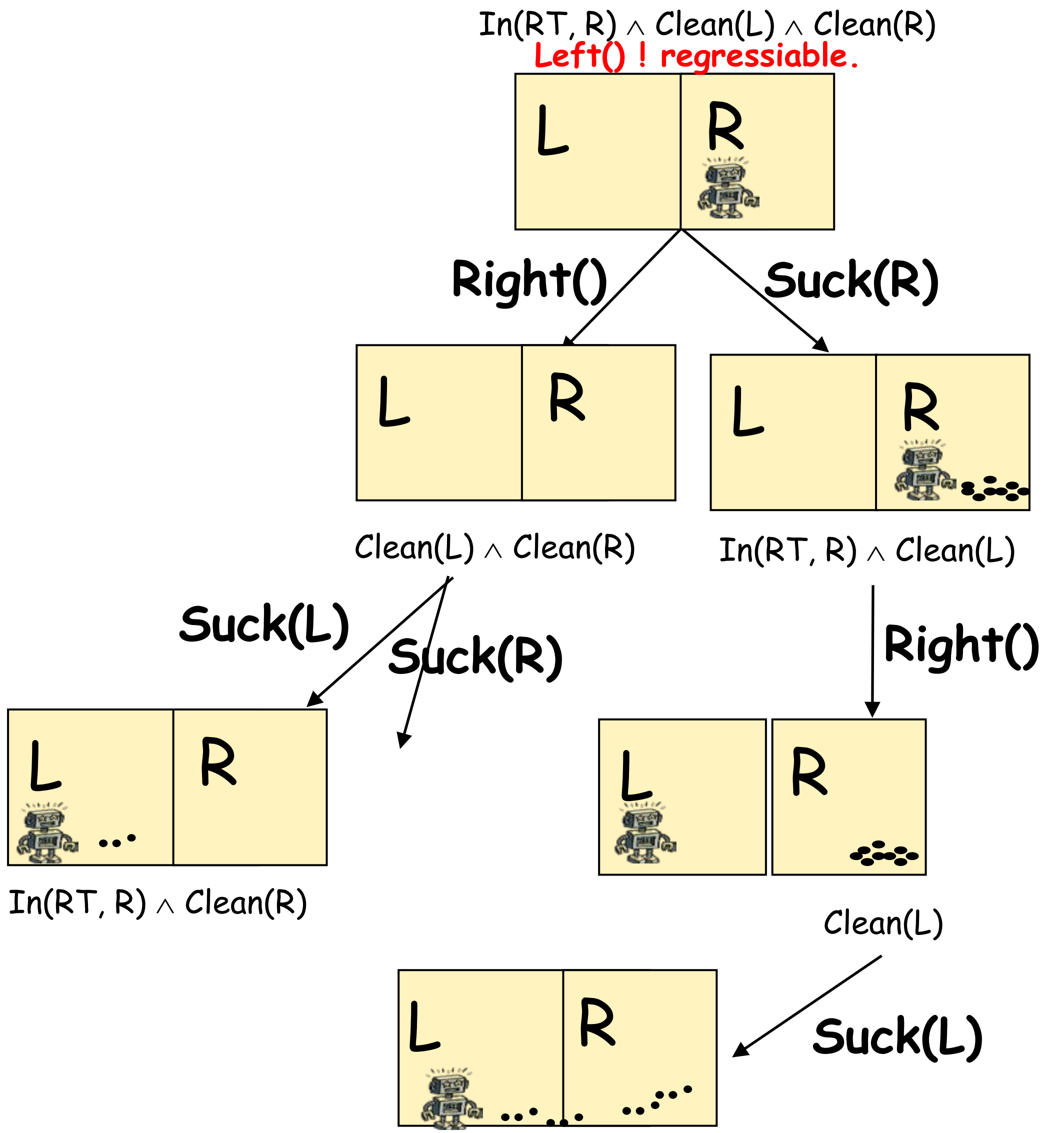
      If SG is not in A's add list then add SG to G'
3. Return G'

Backward Search

Suck(r)
- P = In(Robot, r)
- E = Clean(r)

Left()
- P = ∅
- E = In(Robot, L)
  ¬ In(Robot, R)

Right()
- P = ∅
- E = In(Robot, R)
- ¬ In(Robot, L)



Backward Search – namely known as Means Ends

The *Means* are the available *actions*.

The *Ends* are the *goals* to be achieved.

To solve a list of *Goals* in state *State*, leading to state *FinalState*, do:

If all the *Goals* are true in *State* then *FinalState* = *State*. *Otherwise* do the following:

1. Select a still unsolved *Goal* from *Goals*.
2. Find an *Action* that adds *Goal* to the current state.
3. Enable *Action* by solving the preconditions of *Action*, giving *MidState*.
4. *MidState* is then added as a new *Goal* to *Goals* and the program recurses to step 1.

i.e. we search backwards from the Goal state, generating new states from the preconditions of actions, and checking to see if these are facts in our initial state.

Backward Search – namely known as Means Ends

Means Ends Analysis will usually lead straight from the Goal State to the Initial State as the branching factor of the search space is usually larger going forwards compared to backwards.

However, more complex problems can contain operators with overlapping Add Lists so the MEA would be required to choose between them.

It would require *heuristics*.

Also, linear planners like these will blindly pursue sub-goals without considering whether the changes they are making undermine future goals.

they need someway of *protecting their goals*.

Backward Search – Algorithm

Backward-search(Σ, s0, g0)
$\pi \leftarrow$ hi; g $\leftarrow$ g0
loop
  if s0 satisfies g then return π
  A0 $\leftarrow$ {a $\in$ A | a is relevant for g}
  if A0 = $\emptyset$ then return failure
  non-deterministically choose a $\in$ A0
  g $\leftarrow$ γ−1(g, a)
  $\pi \leftarrow$ a.π

Considering an action a to be relevant for achieving a goal g if
1- action a does not make any of the conditions in g false .. and also ..
2- action a makes at least one the conditions in g true.

Backward Search

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

↓ Stack(C,B)

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

↓ Pickup(C)

On(B,A) $\wedge$ Clear(B) $\wedge$ Handempty $\wedge$ Clear(C) $\wedge$ On(C,Table)

↓ Stack(B,A)

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

↓ Pickup(B)

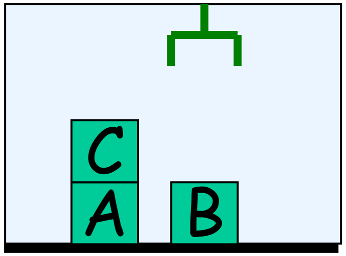
Clear(C) $\wedge$ On(C,Table) $\wedge$ Clear(A) $\wedge$ Handempty $\wedge$ Clear(B) $\wedge$ On(B,Table)

↓ Putdown(C)

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

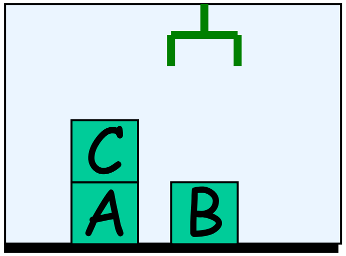
↓ Unstack(C,A)

Clear(B) $\wedge$ On(B,Table) $\wedge$ Clear(C) $\wedge$ Handempty $\wedge$ On(C,A)



Initial state

Backward Search



Initial state

Goal: 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 Search – 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 h 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

Thank you for your attention!

Questions?