POLITECNICO
MILANO 1863

based on Manuela M. Veloso lectures on

PLANNING, EXECUTION AND LEARNING

Cognitive Robotics

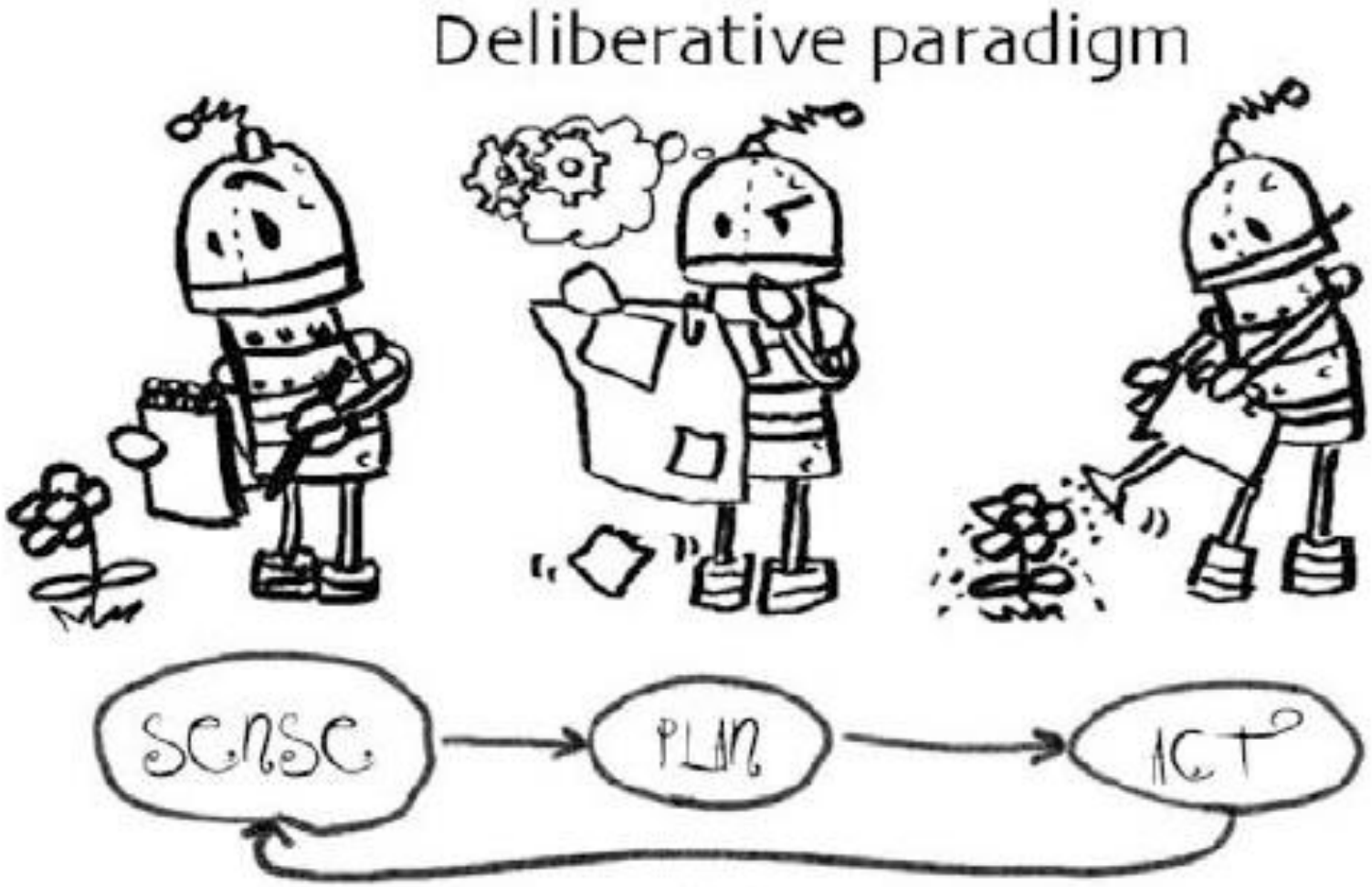
2017/2018

Planning: State, Actions and Goal Representation

Matteo Matteucci
matteo.matteucci@polimi.it

Artificial Intelligence and Robotics Lab - Politecnico di Milano

Recall: «Think hard, act later»



Recall: «Think hard, act later»

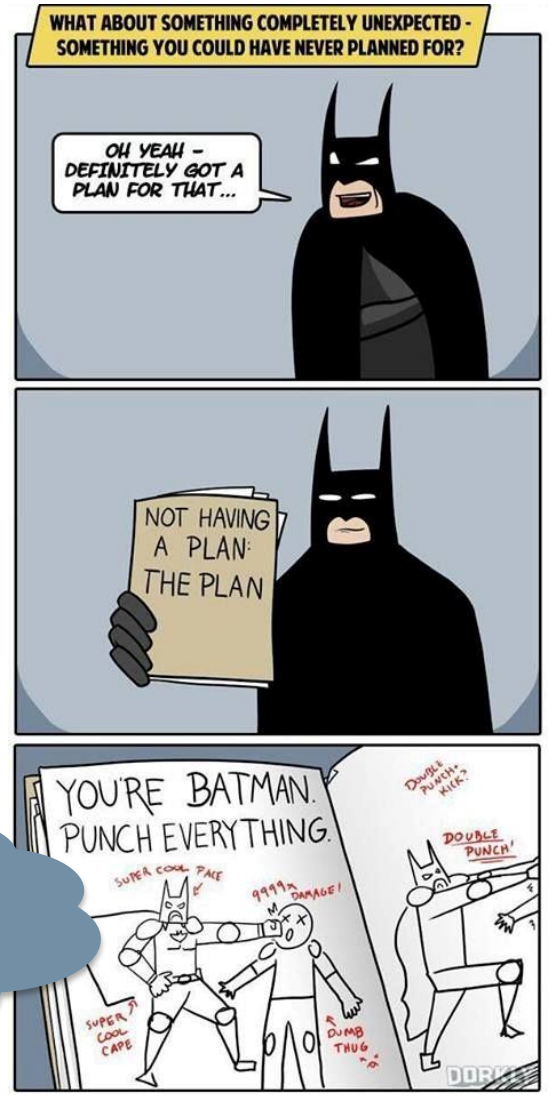
Planning is about «thinking»

- Given the ***actions*** available in a task domain.
- Given a problem specified as:
  - an ***initial state*** of the world
  - a goal statement (***set of goals***) to be achieved
- Find a ***solution*** to the problem

Plan: a way, in terms of a *sequence of actions*, to transform the *initial state* into a *new state* of the world where the *goal statement is true*.

It's all about states, actions, and plans!

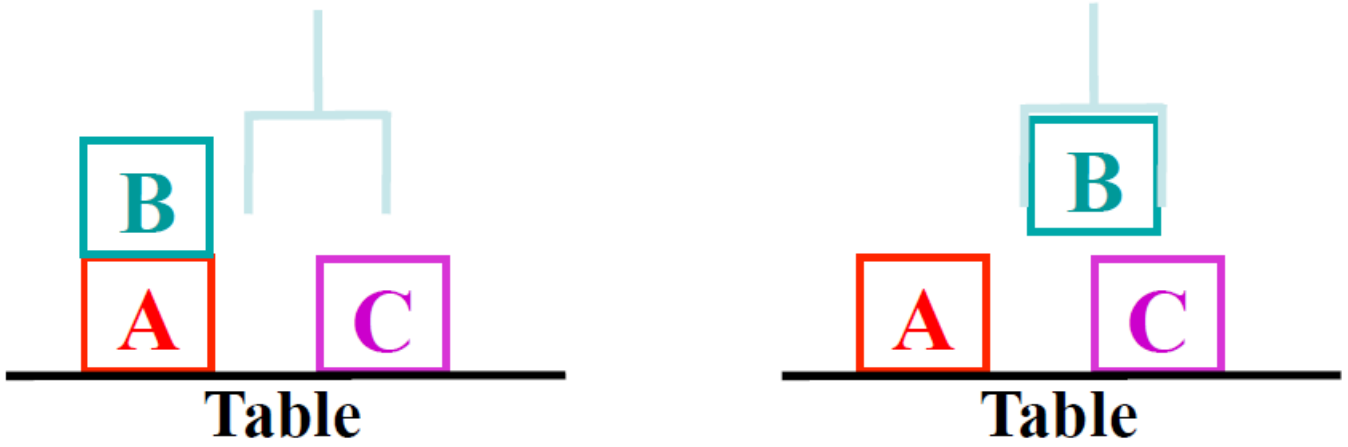
Newell and Simon 1956



The Block World

The Block World is a useful abstraction to introduce *States, Actions and Plans*

- Blocks are on the Table, or on top of each other.
- There is an Arm – the Arm can be empty or holding one block.
- The table is always clear.



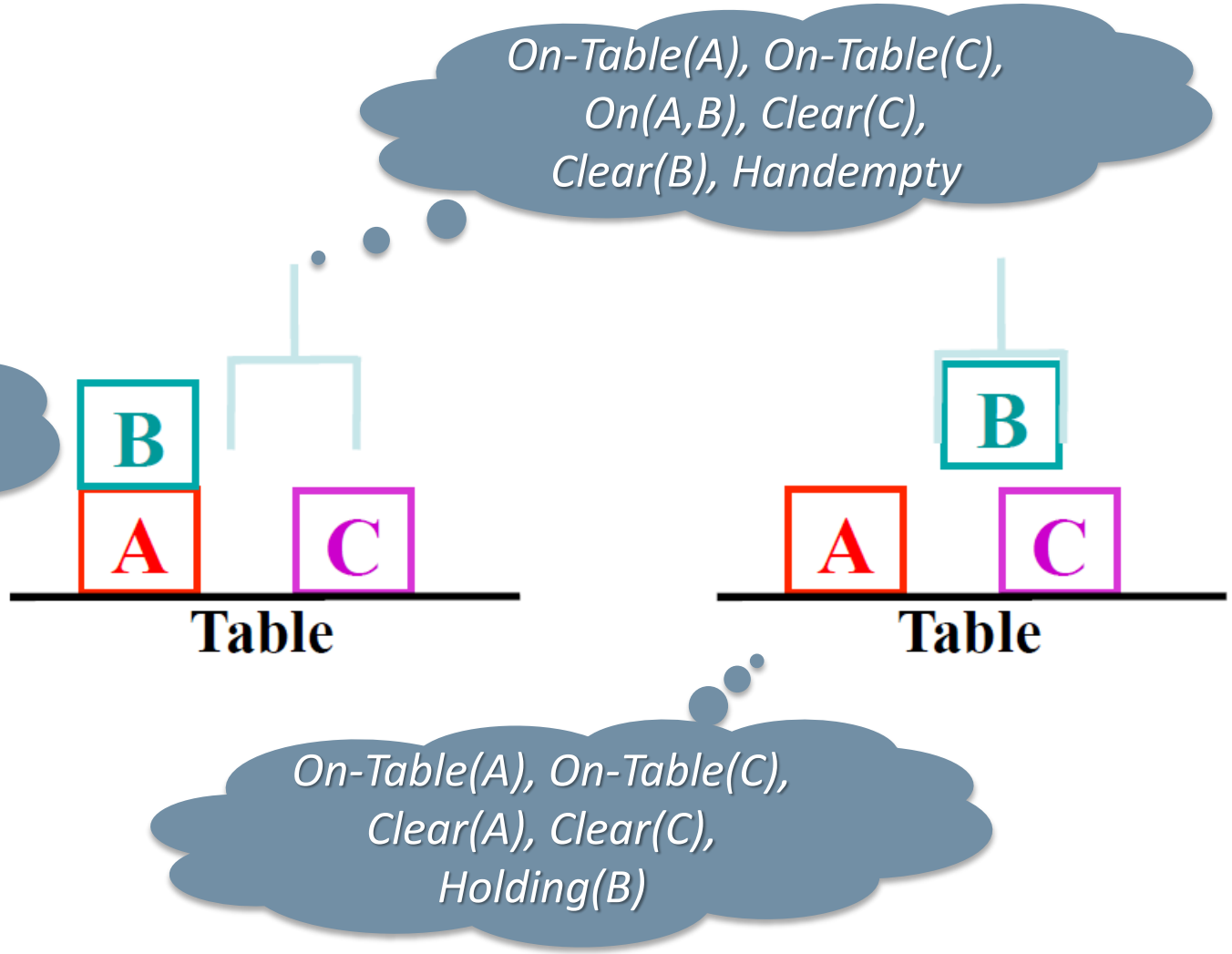
The Block World: States

Objects

- Blocks: *A, B, C*
- Table: *Table*

Predicates

- *On(B, A), On(C, Table)*
- *Clear(B), Handempty, Holding(C)*
- *On-table(A), On(A,B), Top(B),…*



States – Conjunctive

- On(B,A) and On(C,Table) and Clear(B) and Handempty
- …

The Block World: Assumptions/Limitations

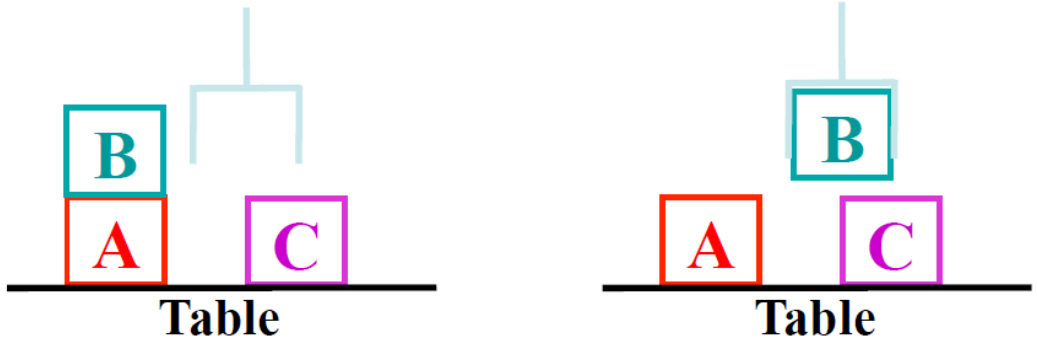
The Block World models Classical Deterministic Planning ...

- There is a single initial state
- The description is complete
- The plan is deterministic
- What is not true in the state is false

CWA: Closed World Assumption

The basic operators perform queries on states

- On(A,B) → returns true or false
- On(A,x) → returns x=Table or x=B
- On-table (x) → returns x=A and x=C
- …



The Block World: State Description with Two Blocks

A-on-B
A-on-Table
B-on-A
B-on-Table
Holding-A ¬*A-on-B* ∧ ¬*A-on-Table*
Holding-B ¬*B-on-A* ∧ ¬*B-on-Table*
Handempty ¬*Holding-A* ∧ ¬*Holding-B*
Clear-A ¬*B-on-A*
Clear-B ¬*A-on-B*

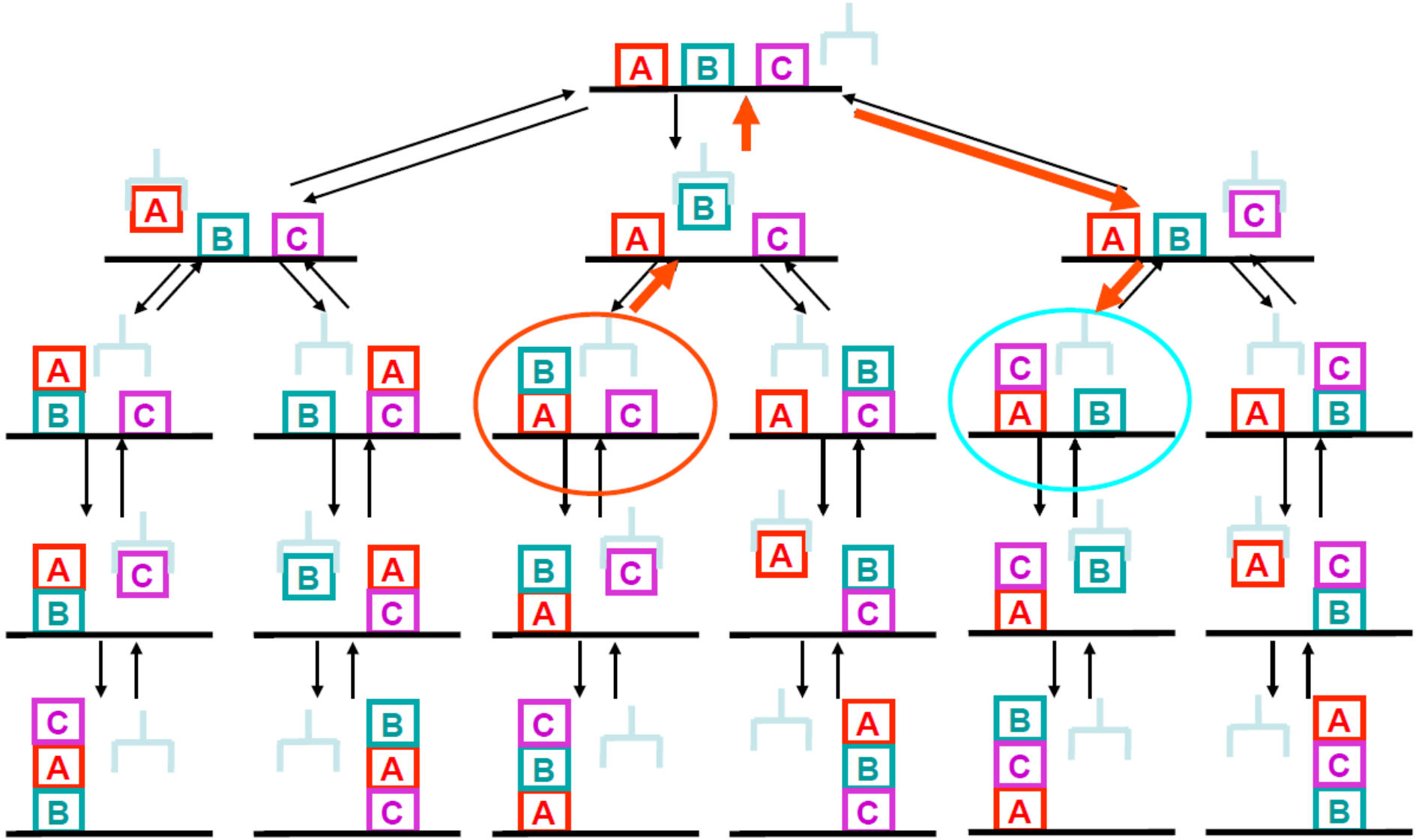
2^4 Possible states

All these define the State Space

A-on-x {∅, table, B}
B-on-x {∅, table, A}

3^2 Possible States

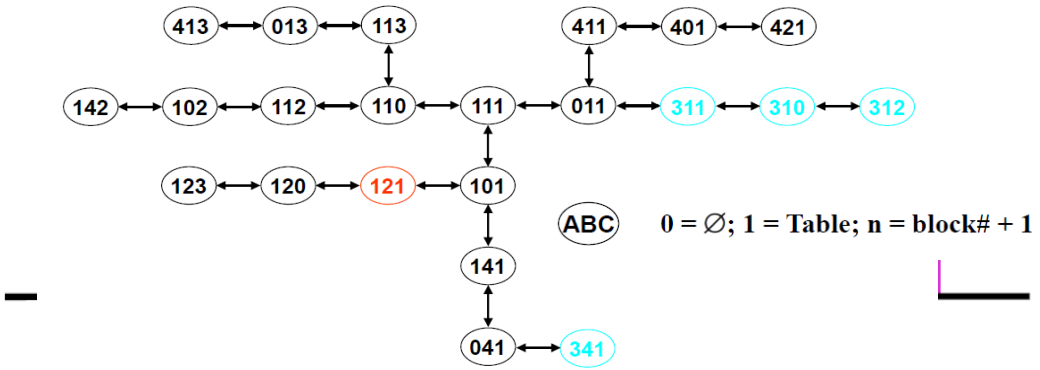
The Block World: Planning as State-Space Search



Different Models for State Spaces

Different models for states exist ...

- Atomic identification of states (s1, s2,...)
- Symbolic feature based states
- Symbolic predicate based states
- …



… together with different ways of combining them

- Conjunctive → observable
- Probabilistic → approximate
- Incremental → on-demand
- Temporal → dynamic

Predicates, conjunctive, complete, correct, deterministic

Goal Specification

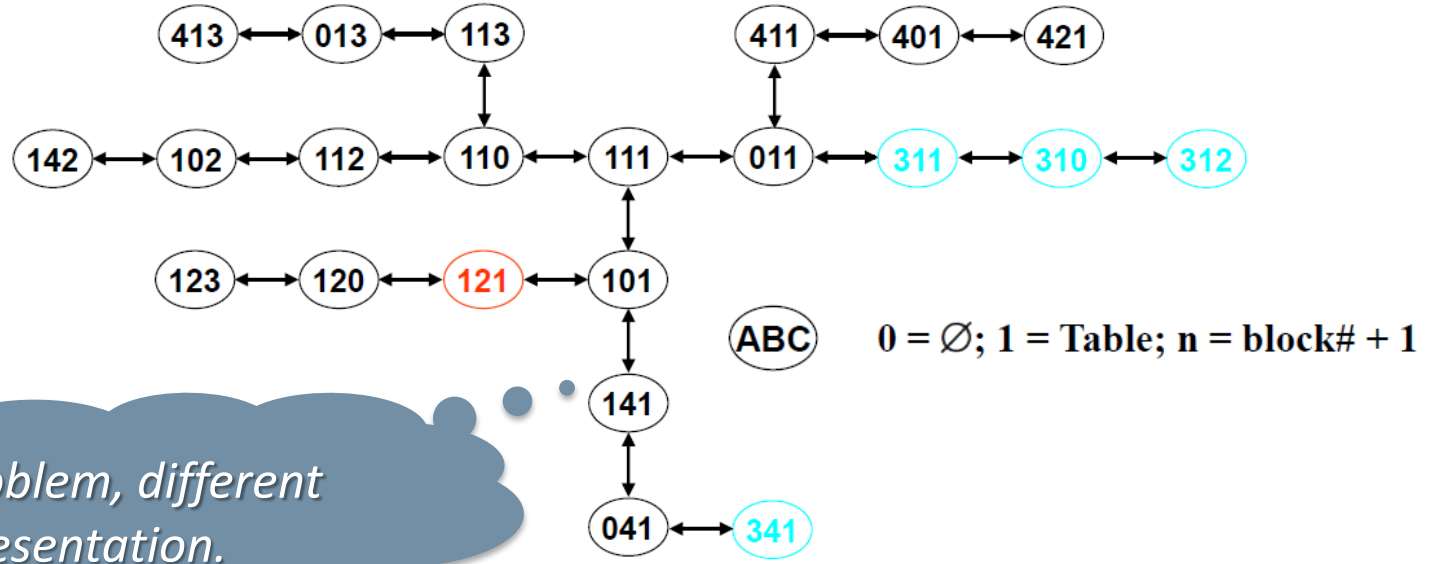
We can specify a Goal according to different levels of generality:

- Goal State → Completely specified state
- Goal Statement → Partially specified state
- Objective function → Defines “good” or “optimal” plan

Increased Generality

Goal Statement example:

- Initial: A-on-x = Table;
  B-on-x = A;
  C-on-x = Table
- Goal: A-on-x = B



Same problem, different representation.

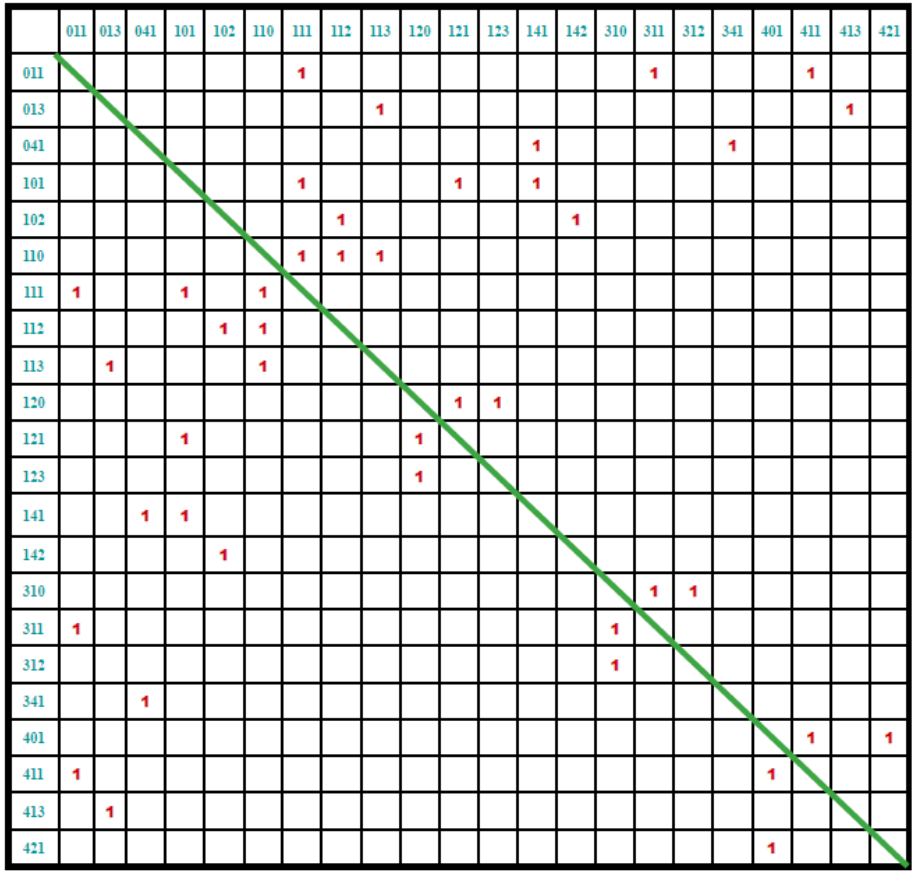
What is an Action?

Plan: a way, in terms of a *sequence of actions*, to transform the *initial state* into a *new state* of the world where the *goal statement is true*.

Newell and Simon 1956

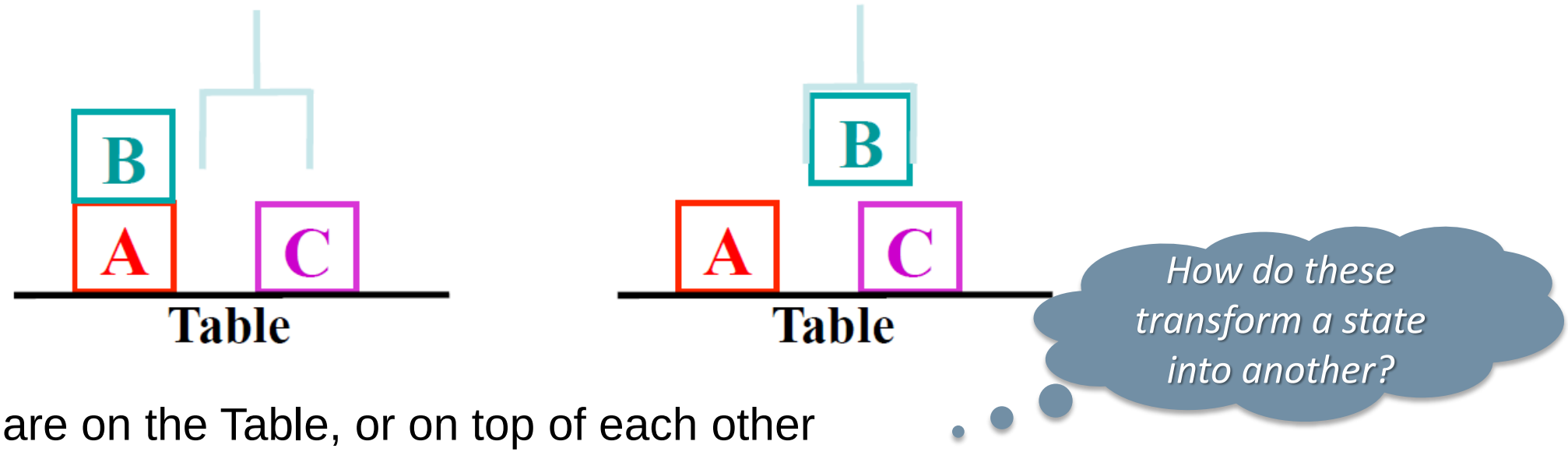
Action: a transition from one (partial) state to another

- May be applicable only in particular states
- Generates new state
  - Deterministic: t_{det}: S x A → S
  - Non-deterministic: $t_{non\text{-}det}$: S x A → 2^S
  - Probabilistic: t_{prob}: S x A → <2^S, r>



Explicit Action Representation

The Block World Dynamics: Actions



- Blocks are on the Table, or on top of each other
- Blocks are picked up and put down by the arm
- A block can be picked up only if it is clear, i.e., without a block on top
- The arm can pick up a block only if the arm is empty, i.e., if it is not holding another block, i.e., the arm can pick up only one block at a time
- The arm can put down blocks on blocks or on the table
- The table is always clear

STRIPS Action Representation

STRIPS (Stanford Research Institute Problem Solver) was the planner used by Shakley, it was developed at SRI International by Richard Fikes and Nils Nilsson in 1971.

Explicit action a representation

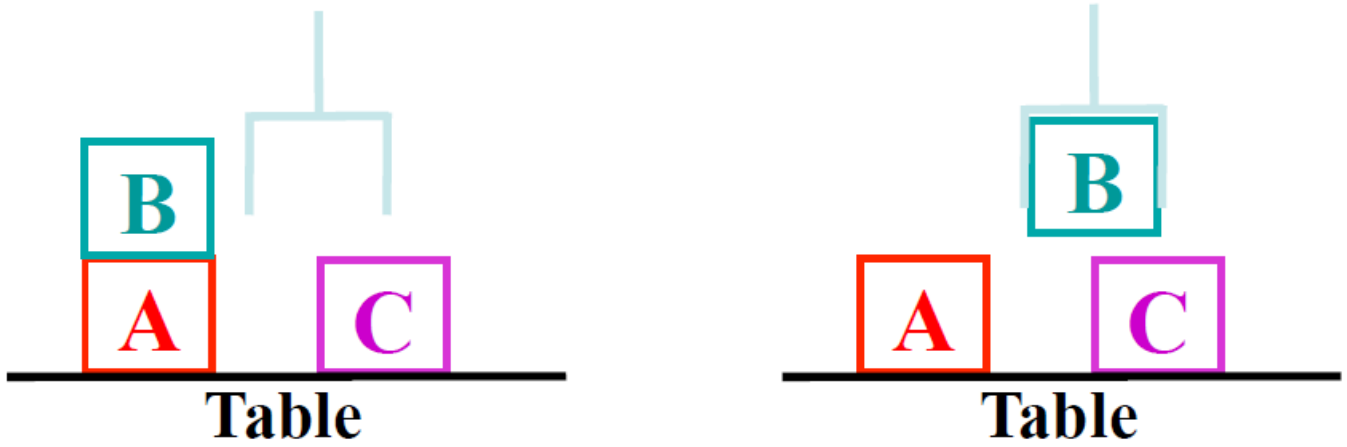
- {preconds(a), effects^{-}(a), effects^{+}(a)}
- effects^{-}(a) $\cap$ effects^{+}(a) = $\emptyset$
- $\tau(\mathcal{S}, a) = \{\mathcal{S} - \text{effects}^{-}(a) \cup \text{effects}^{+}(a)\}$, where $\mathcal{S} \in 2^{S}$

Example in the Block World

- Pickup_from_table(?b)
  Pre: ...
  Add: ...
  Delete: ...

Let's try this out together!

Actions in the Block World



In the Block World:

- An action *a* is **applicable** in *s* if all its preconditions *are satisfied by s*.
- RESULT*(s,a)* = (*s* – Del (*a*)) U Add (*a*)
- No explicit mention of *time*
  - The precondition always refers to time *t*
  - The effect always refers to time *t+1*

The Block World: Actions

```
Pickup_from_table(b)
    Pre: Block(b), Handempty
         Clear(b), On(b,Table)
    Add: Holding(b)
    Delete: Handempty, On(b,Table)
            Clear(b)
```

```
Pickup_from_block(b1, b2)
    Pre: Block(b1),Block(b2), Handempty
         Clear(b1), On(b1,b2)
    Add: Holding(b1), Clear(b2)
    Delete: Handempty, On(b1,b2)
            Clear (b1)
```

```
Putdown_on_table(b)
    Pre: Block(b), Holding(b)
    Add: Handempty,
         On(b, Table)
    Delete: Holding(b)
```

```
Putdown_on_block(b1, b2)
    Pre: Block(b1), Holding(b1)
         Block(b2), Clear(b2), b1 ≠ b2
    Add: Handempty, On(b1, b2)
    Delete: Holding(b1), Clear(b2)
```

More Realistic Actions Representations

Conditional Effects

- Pickup (b)
  Pre: Block(b), Handempty, Clear(b), On(b, x)
  Add: Holding(b)
  if (Block(x)) then Clear(x)
  Delete: Handempty, On(b, x)

Quantified Effects

- Move (o, x)
  Pre: At(o, y), At(Robot, y)
  Add: At(o, x), At(Robot, x)
  forall (Object(u)) [if (In(u, o)) then At(u, y)]
  Delete: At(o, y), At(Robot, y),
  forall (Object(u)) [if (In(u, o)) then At(u, y)]

All these extensions can be emulated adding actions!

Disjunctive and Negated Preconditions

- Holding(x) Or Not[Lighter_Than_Air(x)]

More Realistic Actions Representations

Inference Operators / Axioms

- `Clear(x) iff forall(Block(y))[ Not[On(y, x)]]`

Functional effects

- `Move (o, x)`

```
    Pre: At(o, y), At(Robot, y), Fuel(f), f ≥ Fuel_Needed(y, x)
    Add: At(o, x), At(robot, x), Fuel(f – Fuel_Needed(y, x)),
           forall (Object(u)) [ if (In(u, o)) then At(u, y)]
    Delete: At(o, y), At(Robot, y), Fuel(f),
           forall (Object(u)) [ if (In(u, o)) then At(u, y)]
```

Disjunctive Effects

- `Pickup_from_block(b)`

```
    Pre: Block(b), Handempty, Clear(b), On(b, c), Block(c)
C1: Add: Clear(c), Holding(b); Delete: On(b, c), Handempty
C2: Add: Clear(c), On(b, Table); Delete: On(b, c)
C3: Add: ; Delete:
```

These extensions make the planning problem significantly harder

Much harder and you can add probability!!!

POLITECNICO
MILANO 1863

Cognitive Robotics

Planning: Plan Generation

Matteo Matteucci
matteo.matteucci@polimi.it

Artificial Intelligence and Robotics Lab - Politecnico di Milano

Different Plans ...

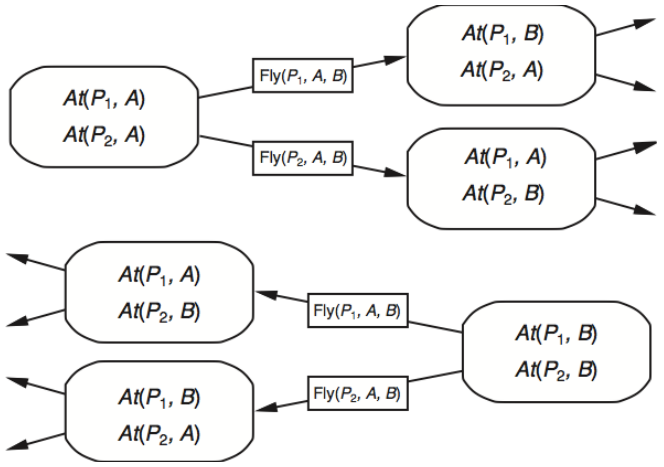
A plan can have different degrees of generality …

- Sequence of Instantiated Actions
- Partial Order of Instantiated Actions
- Set of Instantiated Actions
- Policy (a direct mapping from states to actions)

Increased Generality

… and adopt different search strategies:

- Progression, a.k.a. forward state space search, a.k.a. forward chaining
- Regression, a.k.a. backward state-space search, a.k.a. backward chaining



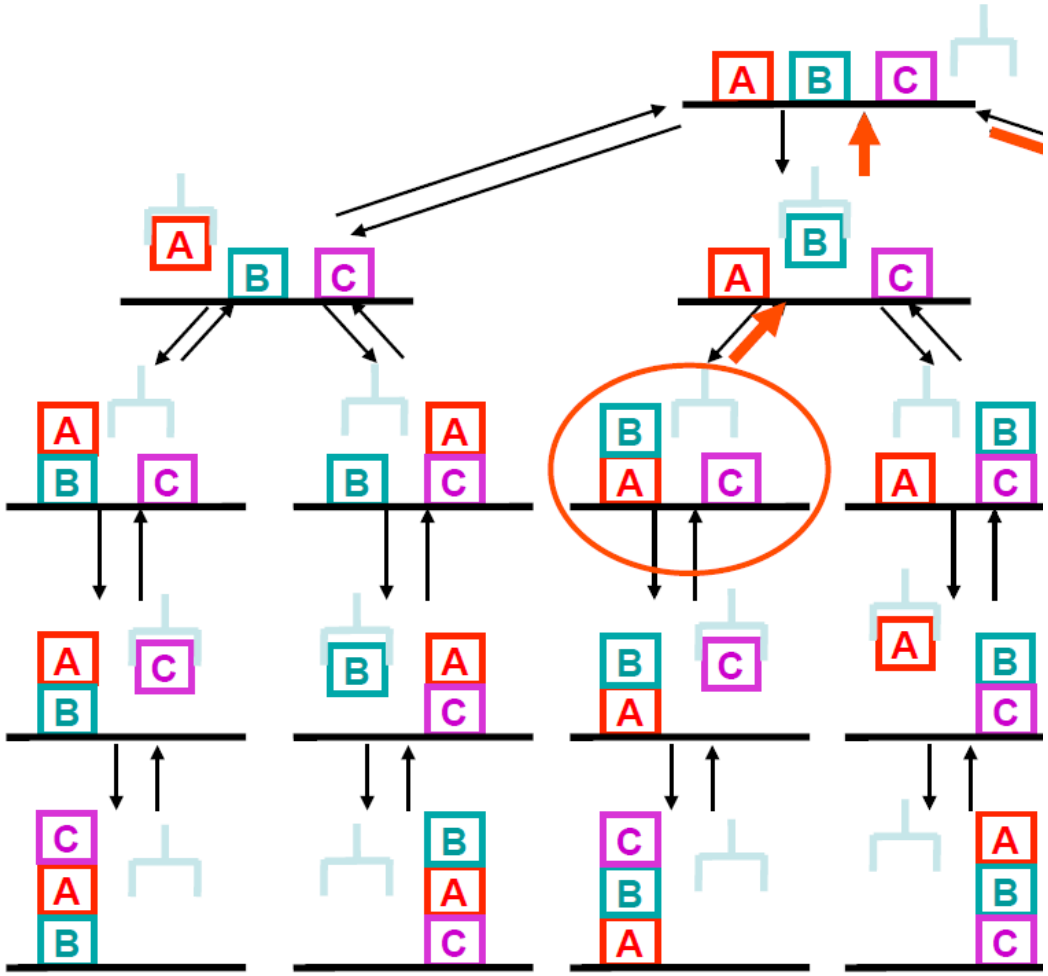
Plan Generation

Backtracking Search Through a Search Space

- How to conduct the search
- How to represent the search space
- How to evaluate the solutions

Non-Deterministic Choices Determine Backtracking

- Choice of actions
- Choice of variable bindings
- Choice of temporal orderings
- Choice of subgoals to work on



Properties of Planning Algorithms

Soundness

- A planning algorithm is ***sound*** if all solutions are legal plans, i.e., all preconditions, goals, and any additional constraints are satisfied

Completeness

- A planning algorithm is ***complete*** if a solution can be found whenever one actually exists
- A planning algorithm is ***strictly complete*** if all solutions are included in the search space

Optimality

- A planning algorithm is ***optimal*** if it maximizes a predefined measure of plan quality

Linear Planning and Means-ends Analysis

Linear Planning

- Uses a Goal stack and work on one goal until completely solved before moving on to the next goal

Mean-ends Analysis

- Search by reducing the difference between the state and the goals, i.e., what means (operators) are available to achieve the desired ends (goal)?

Newell and Simon 60s

```
GPS Algorithm (state, goals, plan)
    If goals ⊆ state, then return (state, plan)
    Choose a difference d ∈ goals between state and goals
    Choose an operator o to reduce the difference d
    If no applicable operators, then return False
    (state, plan) = GPS (state, preconditions(o), plan)
    If state, then return GPS (apply (o, state), goals, [plan,o])

Initial call: GPS (initial-state, initial-goals, [])
```

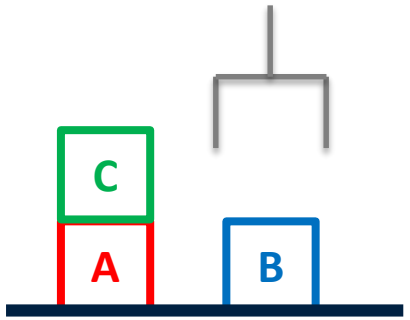
The Block World: GPS at Work

1. Search Stack

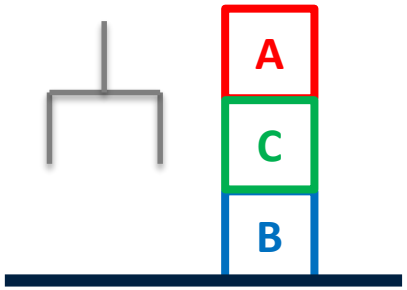
On(A, C) On(C, B)

State

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



State



Goal

2. Search Stack

On(A, C) On(C, B)

On(A, C)

On(C, B)

State

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

3. Search Stack

On(A, C) On(C, B)

On(A, C)

Put_Block(C, B)

Holding(C) Clear(B)

State

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

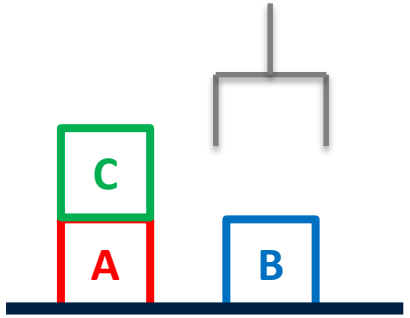
The Block World: GPS at Work

4. Search Stack

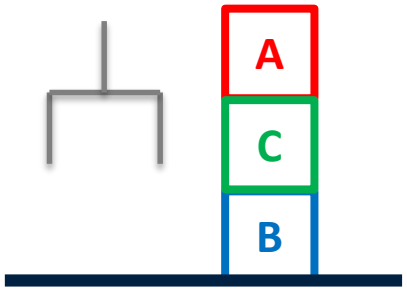
- On(A, C) On(C, B)
- On(A, C)
- Put_Block(C, B)
- Holding(C) Clear(B)
- Holding(C)
- Clear(B)

State

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



State



Goal

5. Search Stack

- On(A, C) On(C, B)
- On(A, C)
- Put_Block(C, B)
- Holding(C) Clear(B)
- Holding(C)

State

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

6. Search Stack

- On(A, C) On(C, B)
- On(A, C)
- Put_Block(C, B)
- Holding(C) Clear(B)
- Pick_Block(C)
- Handempty Clear(C) On(C, ?b)

State

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

The Block World: GPS at Work

7. Search Stack

On(A, C) On(C, B)

On(A, C)

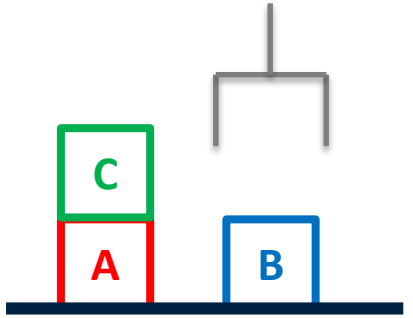
Put_Block(C, B)

Holding(C) Clear(B)

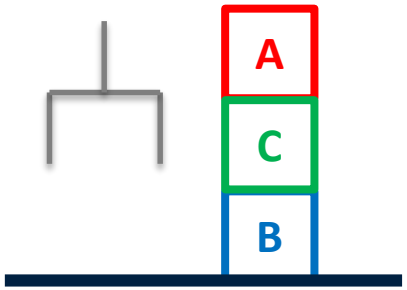
Pick_Block(C)

State

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



State



Goal

The Block World: GPS at Work

7. Search Stack

On(A, C) On(C, B)

On(A, C)

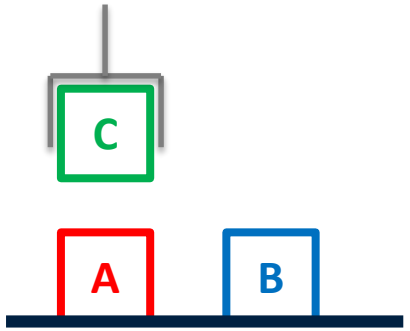
Put_Block(C, B)

Holding(C) Clear(B)

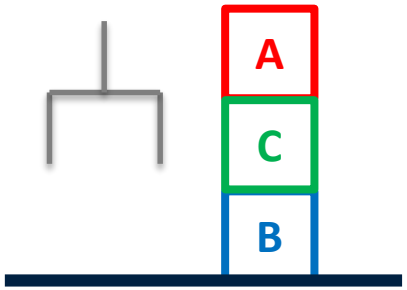
Pick_Block(C)

State

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



State



Goal

8. Search Stack

On(A, C) On(C, B)

On(A, C)

Put_Block(C, B)

Holding(C) Clear(B)

State

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

[Pick_Block(C)]

9. Search Stack

On(A, C) On(C, B)

On(A, C)

Put_Block(C, B)

State

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

[Pick_Block(C)]

The Block World: GPS at Work

10. Search Stack

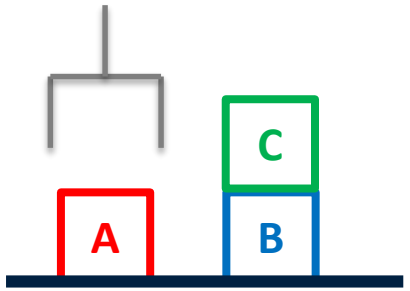
On(A, C) On(C, B)

On(A, C)

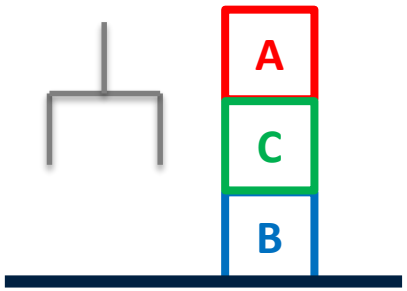
State

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

[Pick_Block(C); Put_Block(C, B)]



State



Goal

11. Search Stack

On(A, C) On(C, B)

Put_Block(A, C)

Holding(A) Clear(C)

State

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

[Pick_Block(C)
Put_Block(C, B)]

12. Search Stack

On(A, C) On(C, B)

Put_Block(A, C)

Holding(A) Clear(C)

Holding(A)

Clear(C)

State

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

[Pick_Block(C)
Put_Block(C, B)]

The Block World: GPS at Work

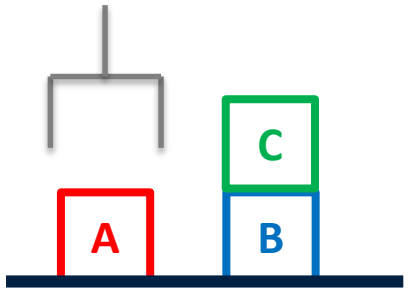
13. Search Stack

- On(A, C) On(C, B)
- Put_Block(A, C)
- Holding(A) Clear(C)
- Holding(A)

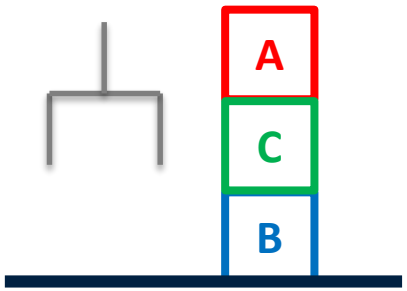
State

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

[Pick_Block(C); Put_Block(C, B)]



State



Goal

14. Search Stack

- On(A, C) On(C, B)
- Put_Block(A, C)
- Holding(A) Clear(C)
- Pick_Table(A)
- Handempty Clear(A) On(A, Table)

State

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

[Pick_Block(C); Put_Block(C, B)]

15. Search Stack

- On(A, C) On(C, B)
- Put_Block(A, C)
- Holding(A) Clear(C)
- Pick_Table(A)

State

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

[Pick_Block(C); Put_Block(C, B)]

The Block World: GPS at Work

16. Search Stack

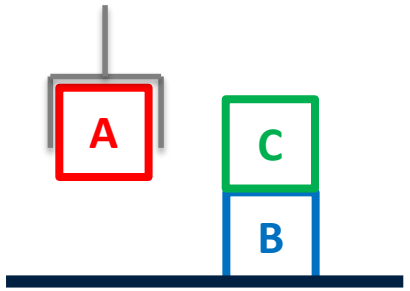
On(A, C) On(C, B)

Put_Block(A, C)

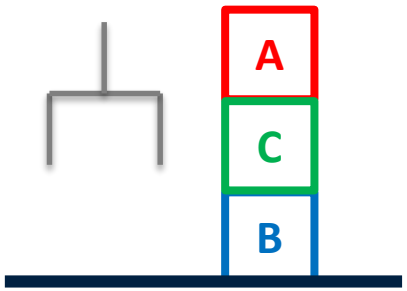
Holding(A) Clear(C)

State

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



State



Goal

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A)]

17. Search Stack

On(A, C) On(C, B)

Put_Block(A, C)

State

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

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A)]

The Block World: GPS at Work

16. Search Stack

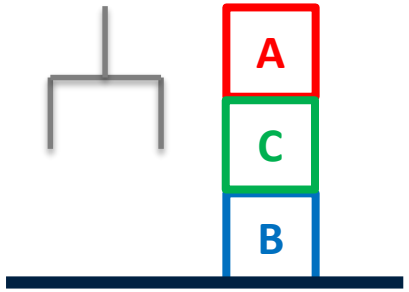
On(A, C) On(C, B)

Put_Block(A, C)

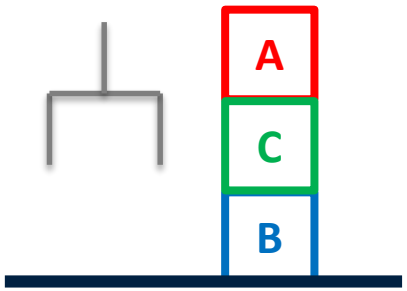
Holding(A) Clear(C)

State

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



State



Goal

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A)]

17. Search Stack

On(A, C) On(C, B)

Put_Block(A, C)

State

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

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A)]

18. Search Stack

On(A, C) On(C, B)

State

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

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A);
Put_Block(A, C)]

The Block World: GPS at Work

Sound? Optimal? Complete?

16. Search Stack

On(A, C) On(C, B)

Put_Block(A, C)

Holding(A) Clear(C)

State

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

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A)]

17. Search Stack

On(A, C) On(C, B)

Put_Block(A, C)

State

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

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A)]

18. Search Stack

On(A, C) On(C, B)

State

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

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A);
Put_Block(A, C)]

19. Search Stack

State

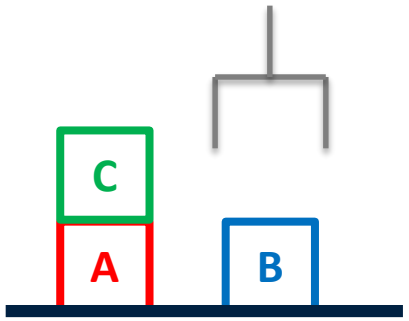
On(B, Table)
Clear(A)
On(C, B)
Handempty
On(A, C)

[Pick_Block(C);
Put_Block(C, B);
Pick_Table(A);
Put_Block(A, C)]

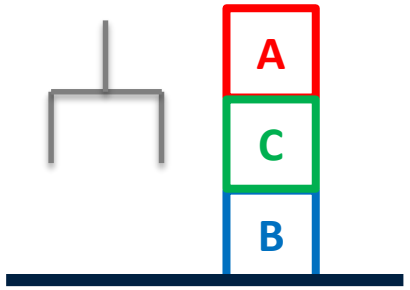
The Sussman Anomaly

Pickup(?b)
Pre: (handempty)
(clear ?b)
(on-table ?b)
Add: (holding ?b)
Delete: (handempty)
(on-table ?b)
(clear ?b)

Putdown(?b)
Pre: (holding ?b)
Add: (handempty)
(on-table ?b)
(clear ?b)
Delete: (holding ?b)



State



Goal

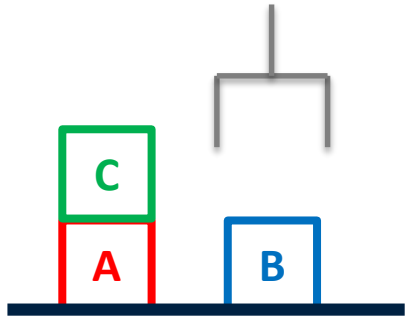
Unstack(?a, ?b)
Pre: (handempty)
(clear ?a) (on ?a ?b)
Add: (holding ?a) (clear ?b)
Delete: (handempty)
(on ?a ?b) (clear ?a)

Stack(?a, ?b)
Pre: (holding ?a)
(clear ?b)
Add: (handempty) (on ?a ?b)
Delete: (holding ?a)
(clear ?b)

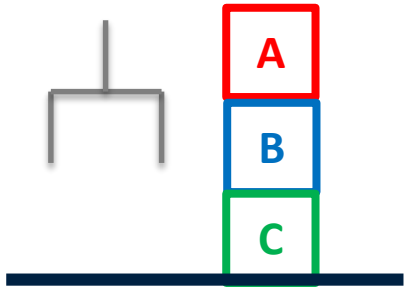
The Sussmann Anomaly – Linear Solution 1

(on B C)

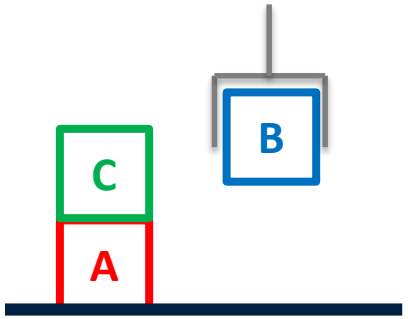
- Pickup (B)
- Stack (B, C)



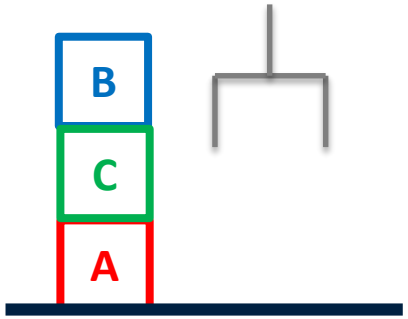
State



Goal



State

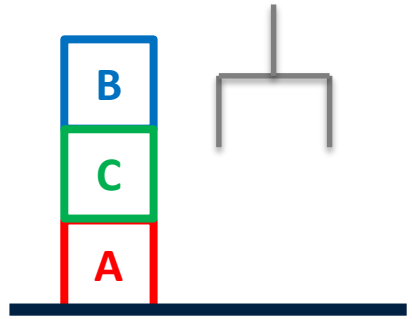


State

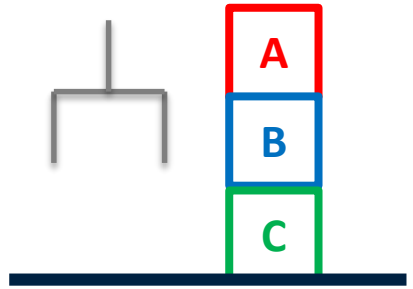
The Sussmann Anomaly – Linear Solution 1

(on B C)
- Pickup (B)
- Stack (B, C)

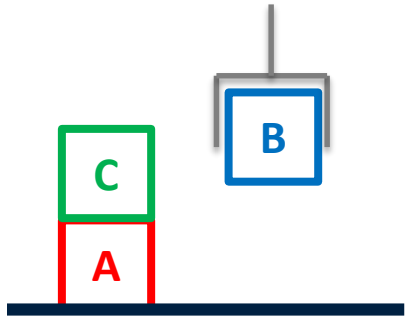
(on A B)
- Unstack (B, C)
- Putdown (B)
- Unstack (C, A)
- Putdown (C)
- Pickup(A)
- Stack (A, B)



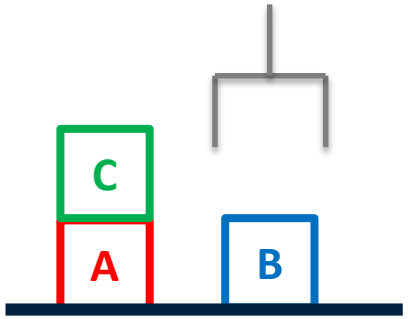
State



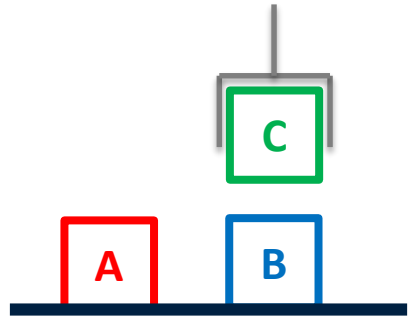
Goal



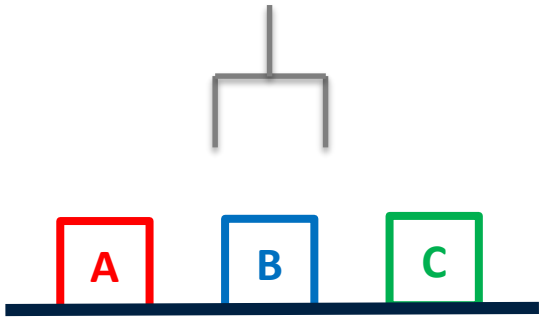
State



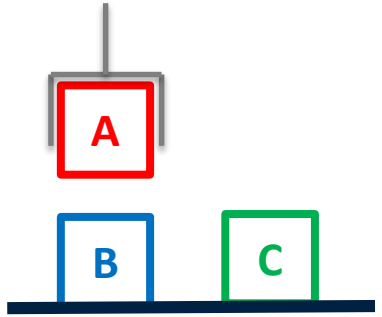
State



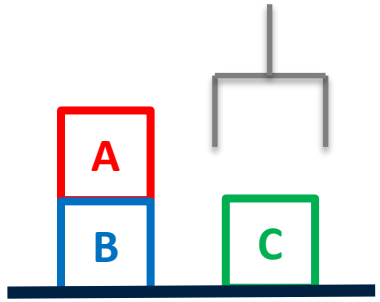
State



State



State



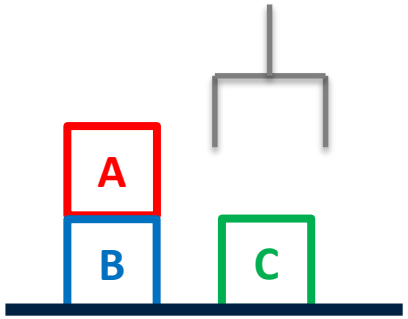
State

The Sussmann Anomaly – Linear Solution 1

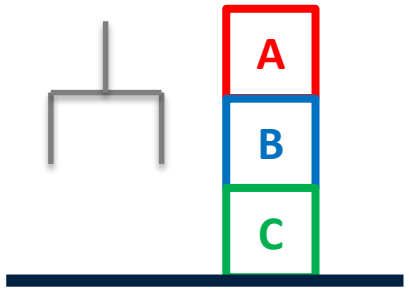
(on B C)
- Pickup (B)
- Stack (B, C)

(on A B)
- Unstack (B, C)
- Putdown (B)
- Unstack (C, A)
- Putdown (C)
- Pickup(A)
- Stack (A, B)

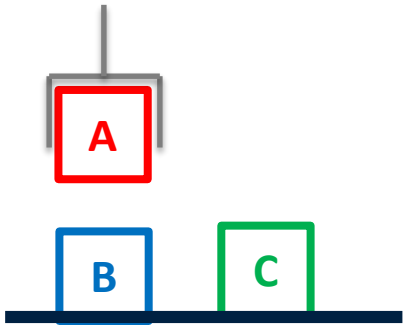
(on B C)
- Unstack (A, B)
- Putdown (A)
- Pickup (B)
- Stack (B, C)



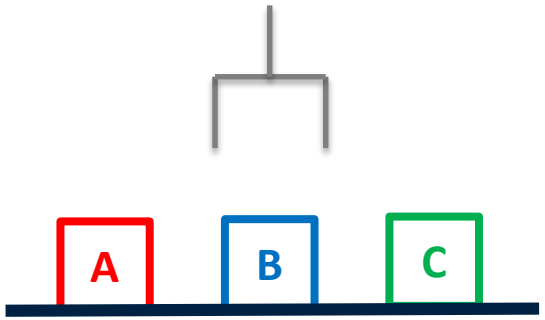
State



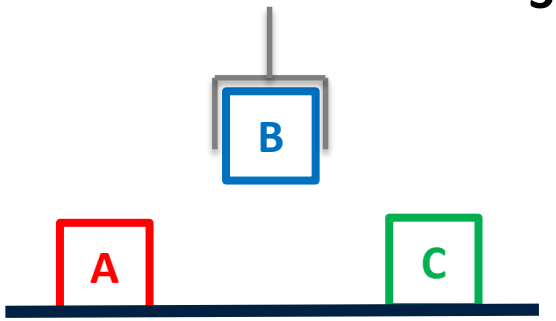
Goal



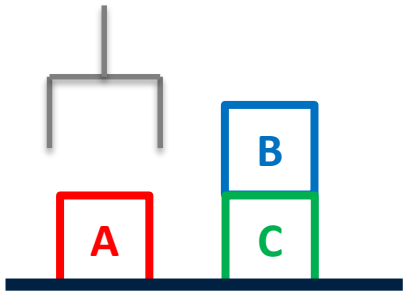
State



State



State



State

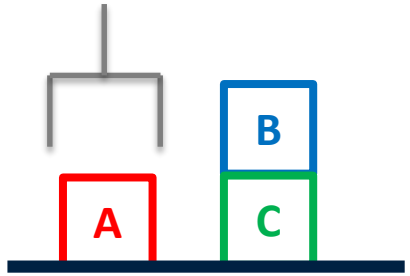
The Sussmann Anomaly – Linear Solution 1

(on B C)
- Pickup (B)
- Stack (B, C)

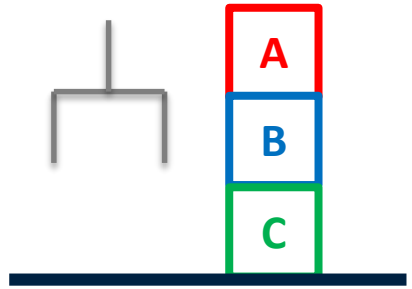
(on A B)
- Unstack (B, C)
- Putdown (B)
- Unstack (C, A)
- Putdown (C)
- Pickup(A)
- Stack (A, B)

(on B C)
- Unstack (A, B)
- Putdown (A)
- Pickup (B)
- Stack (B, C)

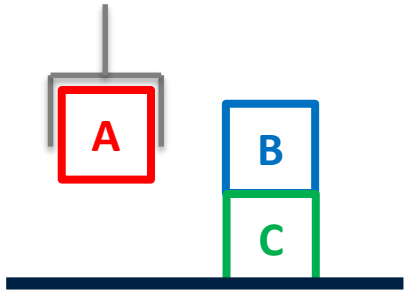
(on A B)
- Pickup (A)
- Stack (A,B)



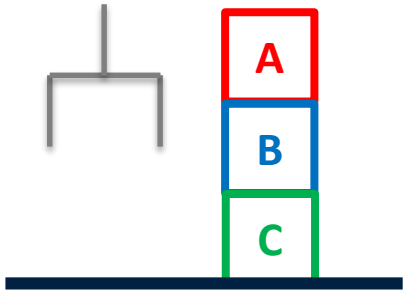
State



Goal



State

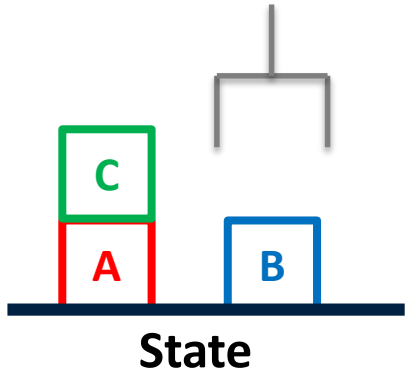


State

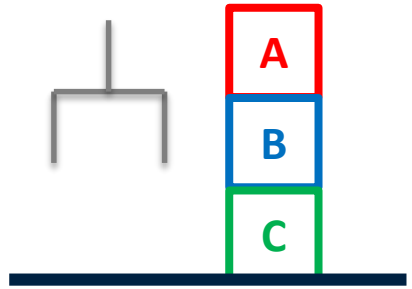
The Sussmann Anomaly – Linear Solution 2

(on A B)

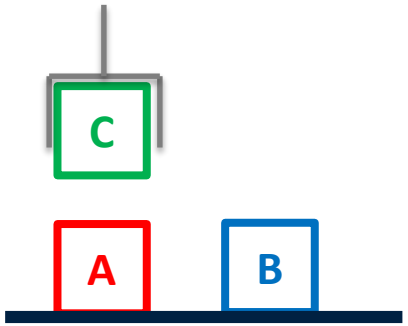
- Unstack (C, A)
- Putdown (C)
- Pickup(A)
- Stack (A, B)



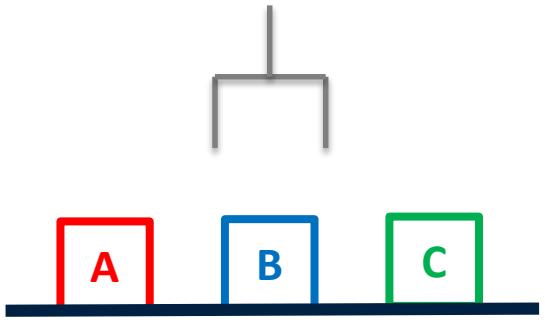
State



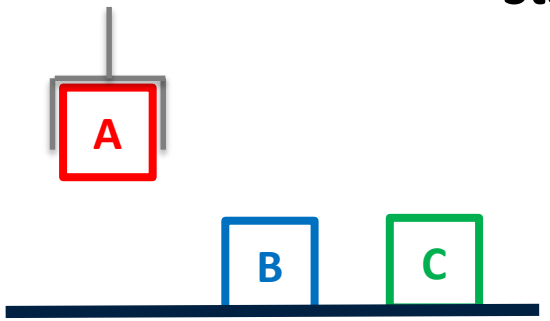
Goal



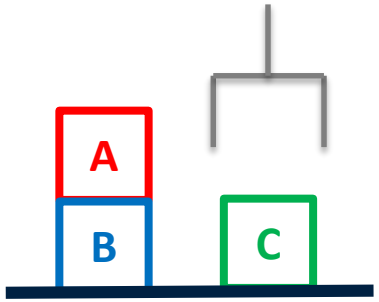
State



State



State

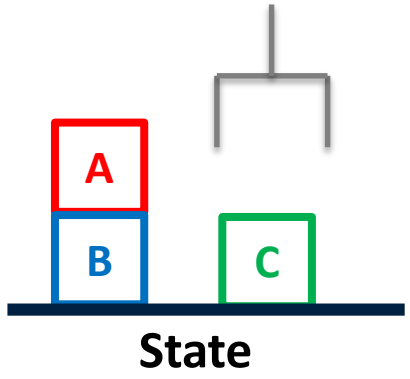


State

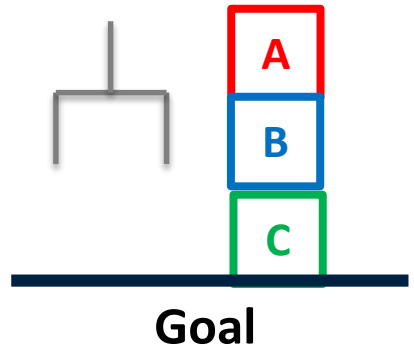
The Sussmann Anomaly – Linear Solution 2

(on A B)
- Unstack (C, A)
- Putdown (C)
- Pickup(A)
- Stack (A, B)

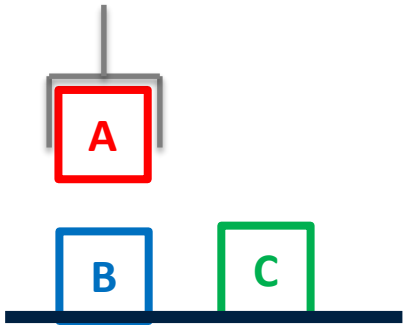
(on B C)
- Unstack (A, B)
- Putdown (A)
- Pickup (B)
- Stack (B, C)



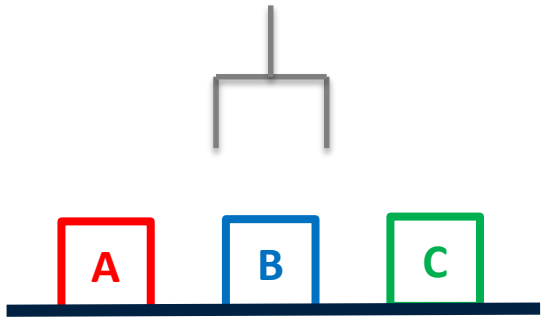
State



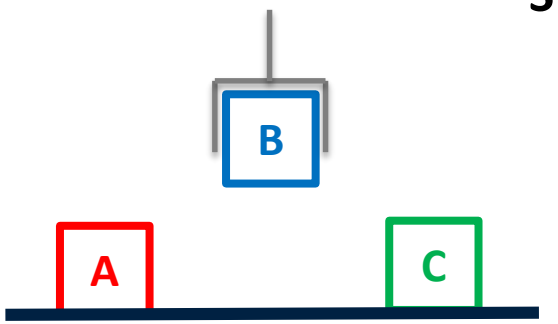
Goal



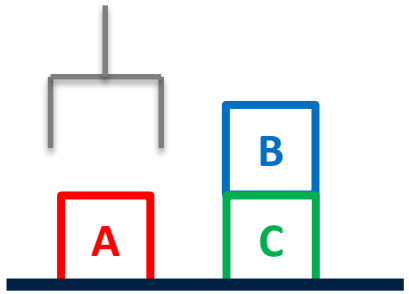
State



State



State



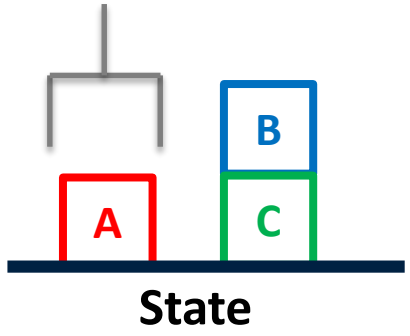
State

The Sussmann Anomaly – Linear Solution 2

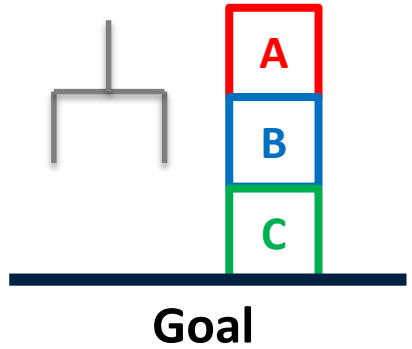
(on A B)
- Unstack (C, A)
- Putdown (C)
- Pickup(A)
- Stack (A, B)

(on B C)
- Unstack (A, B)
- Putdown (A)
- Pickup (B)
- Stack (B, C)

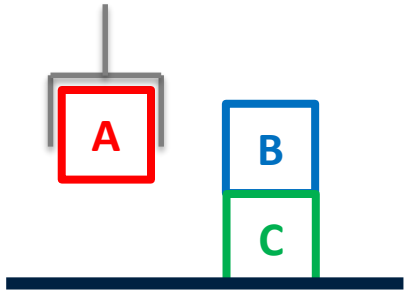
(on A B)
- Pickup (A)
- Stack (A,B)



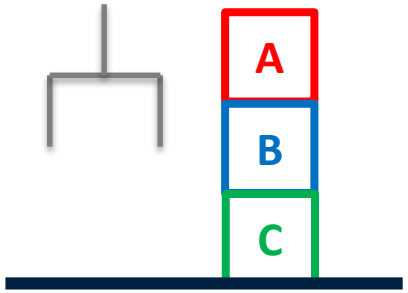
State



Goal



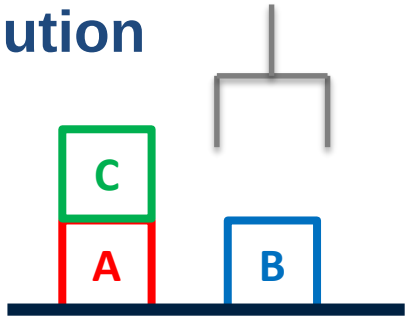
State



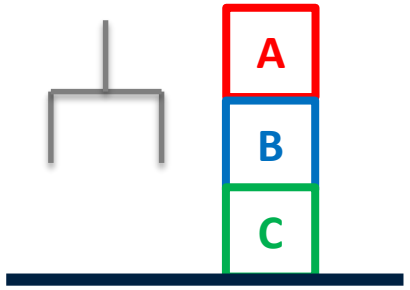
State

Is it Optimal? Can we do it with less actions?

The Sussmann Anomaly: Non Linear (Optimal) Solution

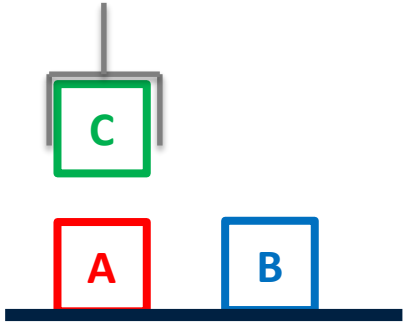


State

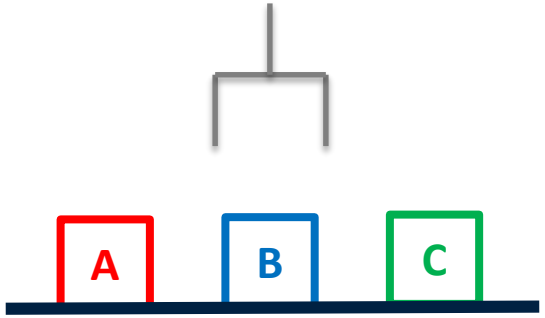


Goal

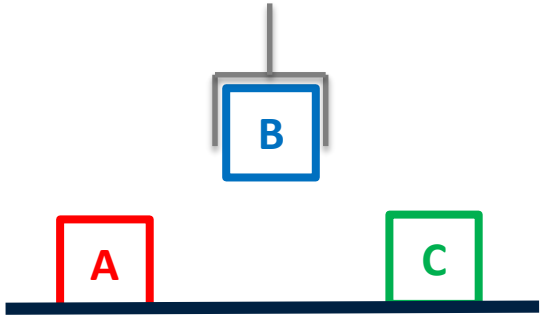
```
(on A B)
    • Unstack (C, A)
    • Putdown (C)
(on B C)
    • Pickup (B)
    • Stack (B, C)
(on A B)
    • Pickup(A)
    • Stack(A, B)
```



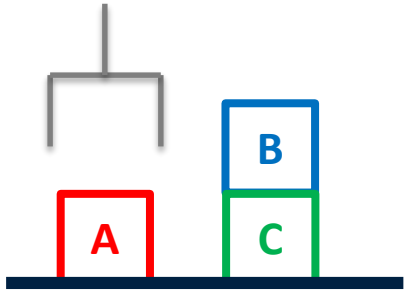
State



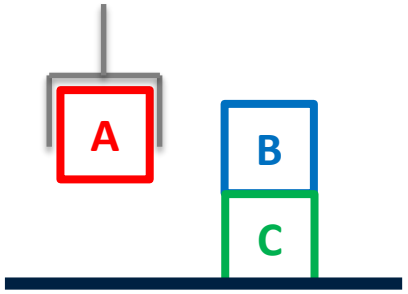
State



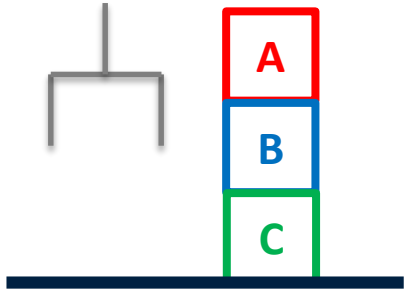
State



State



State



State

Linear Planning and the Goal Stack

Advantages

- Reduced search space, since goals are solved one at a time, and not all possible goal orderings are considered
- Advantageous if goals are (mainly) independent
- Linear planning is sound

Disadvantages

- Linear planning may produce suboptimal solutions (based on the number of operators in the plan)
- Planner's efficiency is sensitive to goal orderings
    - Control knowledge for the “right” ordering
    - Random restarts
    - Iterative deepening

One Way Rocket (Veloso '89)

```
(OPERATOR LOAD-ROCKET
 :preconds
  ?roc ROCKET
  ?obj OBJECT
  ?loc LOCATION
 (and (at ?obj ?loc)
      (at ?roc ?loc))
 :effects
  add (inside ?obj ?roc)
  del (at ?obj ?loc))
```

```
(OPERATOR UNLOAD-ROCKET
 :preconds
  ?roc ROCKET
  ?obj OBJECT
  ?loc LOCATION
 (and (inside ?obj ?roc)
      (at ?roc ?loc))
 :effects
  add (at ?obj ?loc)
  del (inside ?obj ?roc))
```

```
(OPERATOR MOVE-ROCKET
 :preconds
  ?roc ROCKET
  ?from-l LOCATION
  ?to-l LOCATION
 (and (at ?roc ?from-l)
      (has-fuel ?roc))
 :effects
  add (at ?roc ?to-l)
  del (at ?roc ?from-l)
  del (has-fuel ?roc))
```

```
Initial state:
        (at obj1 locA)
        (at obj2 locA)
        (at ROCKET locA)
        (has-fuel ROCKET)
```

```
Goal statement:
        (and
          (at obj1 locB)
          (at obj2 locB))
```

Goal	*Plan*
(at obj1 locB)	(LOAD-ROCKET obj1 locA) (MOVE-ROCKET) (UNLOAD-ROCKET obj1 locB)
(at obj2 locB)	*failure*

State Space Non Linear Planning

Extend linear planning:

- From stack to set of goals
- Include in the search space all possible interleaving of goals

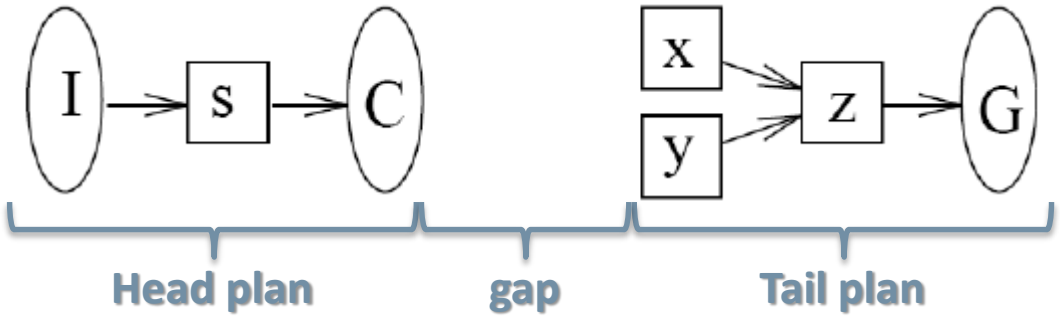
State-space nonlinear planning is complete

Goal	*Plan*
(at obj1 locB)	(LOAD-ROCKET obj1 locA)
(at obj2 locB)	(LOAD-ROCKET obj2 locA)
(at obj1 locB)	(MOVE-ROCKET) (UNLOAD-ROCKET obj1 locB)
(at obj2 locB)	(UNLOAD-ROCKET obj1 locB)

Prodigy4.0 (Veloso et al. 90)

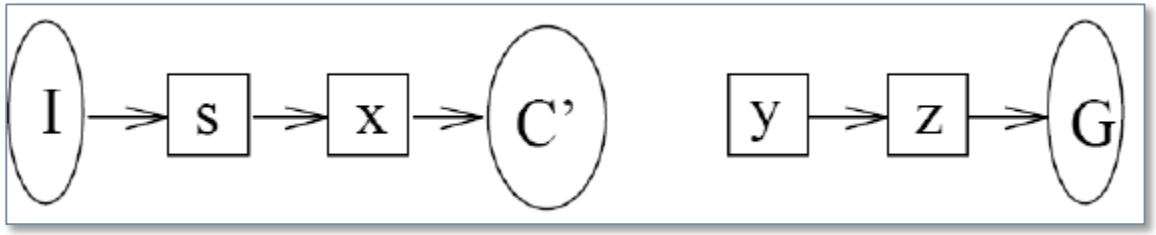
1. Terminate if the goal statement is satisfied in the current state. Initially the set of applicable relevant operators is empty.
2. Compute the SET of pending goals G, and the SET of applicable relevant operators A.
   - A goal is pending if it is a precondition, not satisfied in the current state, of a relevant operator already in the plan.
   - A relevant operator is applicable when all its preconditions are satisfied in the state.
3. Choose a pending goal G in G or choose a relevant applicable operator A in A.
4. If the pending goal G has been chosen, then
   - Expand goal G, i.e., get the set O of relevant instantiated operators that could achieve G,
   - Choose an operator O from O, as a relevant operator for goal G.
   - Go to step 1.
5. If a relevant operator A has been selected as directly applicable, then
   - Apply A,
   - Go to step 1.

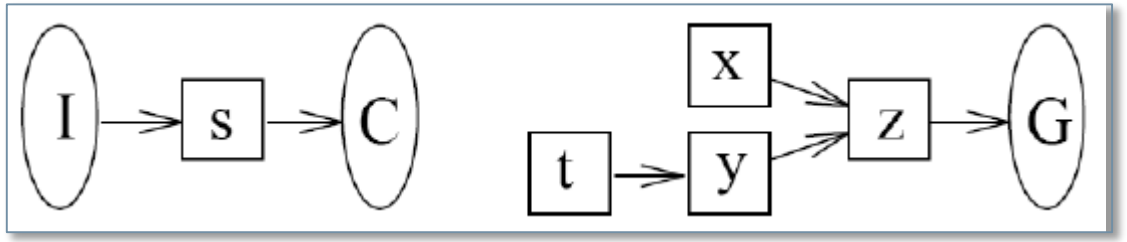
Prodigy4.0 Search Representation



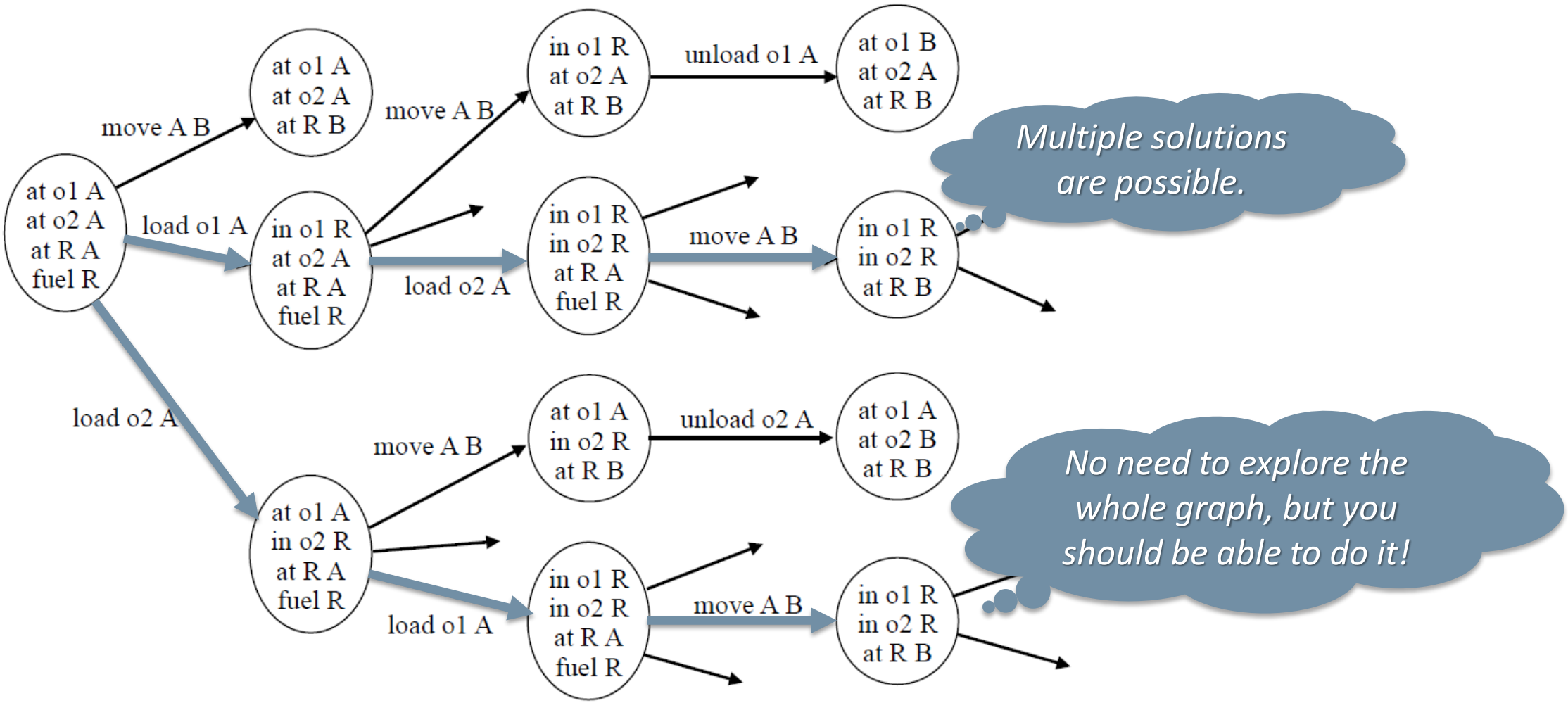
Applying and Operator (moving it to the head)

Adding and operator to the tail plan





After all, it is all about graph exploration



Planning issues

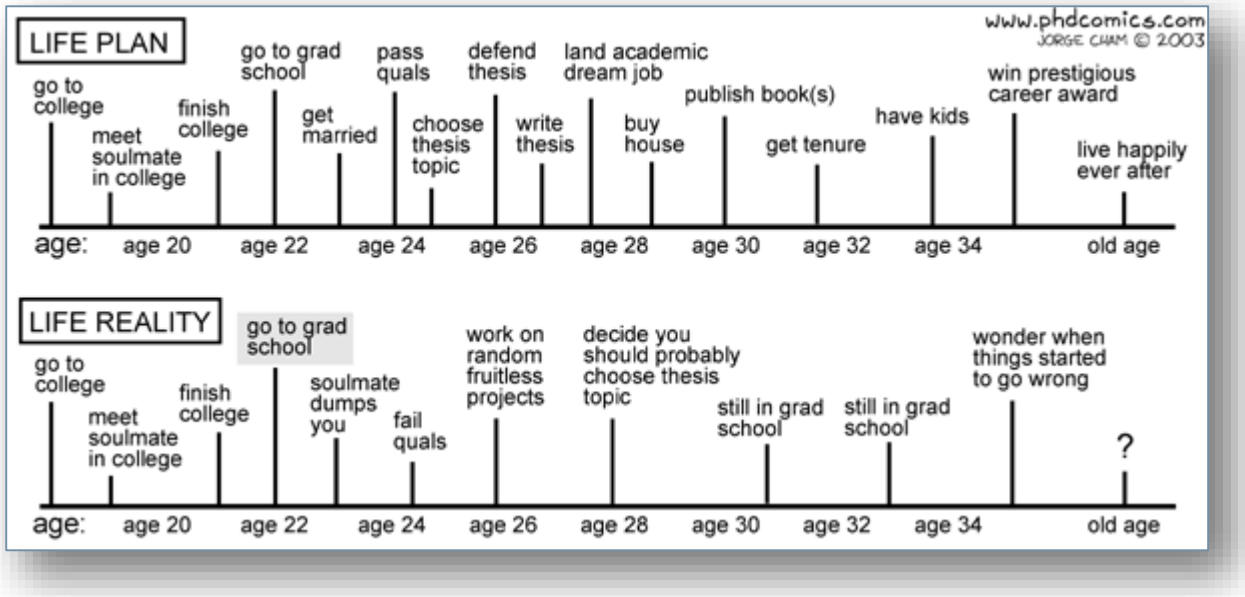
State representation

- The frame problem
- The “choice” of predicates
  (e.g., On-table (x), On (x, table), On-table-A, On-table-B,…)

Action representation

- Many alternative definitions
- Reduce to “needed” definition
- Conditional effects
- Uncertainty
- Quantification
- Functions

Generation – planning algorithm(S)



Wrap-up slide on “Planning and Plan Generation”

What should remain from this lecture?

- Planning: selecting one sequence of actions (operators) that transform (apply to) an initial state to a final state where the goal statement is true.
- Means-ends analysis: identify and reduce, as soon as possible, differences between state and goals.
- Linear planning: backward chaining with means-ends analysis using a stack of goals, potentially efficient, possibly unoptimal, incomplete; GPS
- Nonlinear planning with means-ends analysis: backward chaining using a set of goals; reason about when “to reduce the differences;” Prodigy4.0.

References

- S. Russell, P. Norvig. «Artificial Intelligence: A Modern Approach». Chapter 11: Planning, pages 375-416.Pearson, 2010.

POLITECNICO
MILANO 1863

Cognitive Robotics

Planning: Plan Domain Description Language

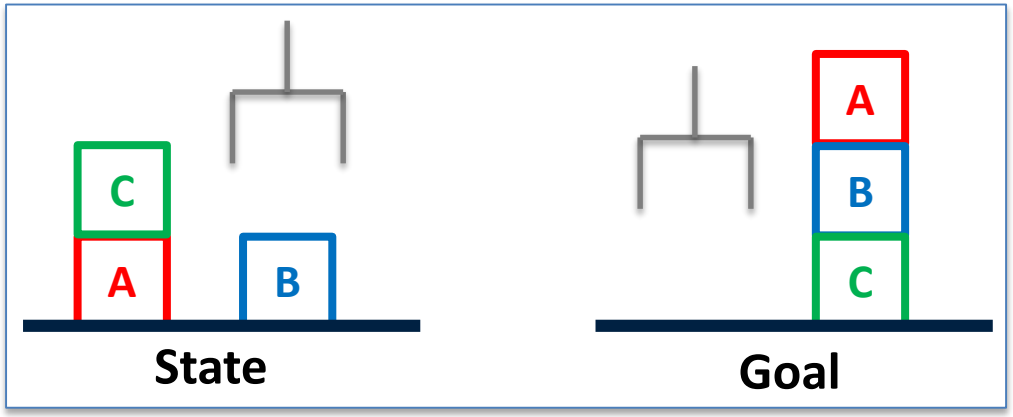
Matteo Matteucci
matteo.matteucci@polimi.it

Artificial Intelligence and Robotics Lab - Politecnico di Milano

Planning Problems in Artificial Intelligence

Planning Problem := <P,A,S,G>

- P:= a SET of Predicates
- A:= a SET of Operators (Actions)
- S:= initial State
- G:= Goal(s)



A Plan Domain or Domain Theory is defined as := P + A

A Problem Solution or Plan is := a sequence of Actions that

- *if executed* from the initial state S
- *will result* in a state satisfying the Goal

STRIPS as a Language

STRIPS has been used as formal language for Planning Problems

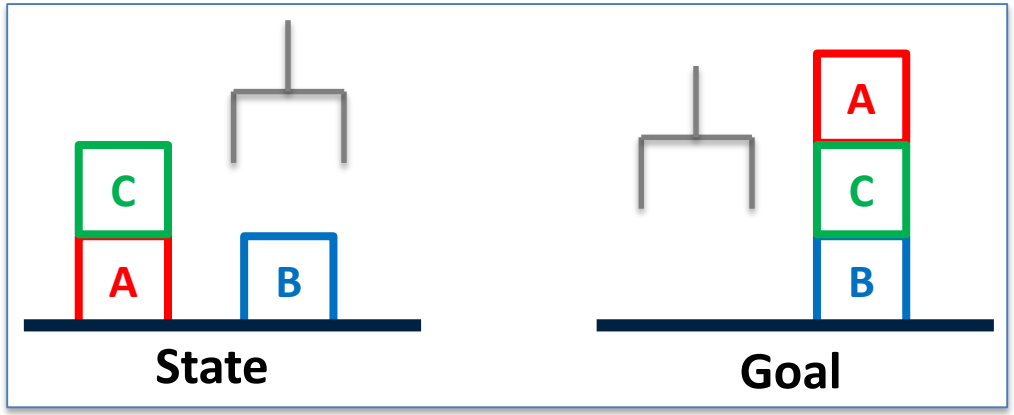
- list of Predicates: *atomic formulae*
- list of Actions:
  - *NAME: string*
  - PRECONDITIONS: *PartiallySpecifiedState*
  - EFFECTS: *ADDlist, DELETElist*
  - + *"STRIPS assumption"*
- Initial State: *State*
- Goal: *PartiallySpecifiedState*

"A State S satisfies a PartiallySpecifiedState G if S contains all the atoms of G"

- Atomic formula (atom):= predicate + arguments
- State:= set of positive atoms + CWA!
- PartiallySpecifiedState:= set of positive atoms

The Block World in STRIPS

- *empty*: the gripper is not holding a block
- *holding(B)*: the gripper is holding block B
- *on(B1,B2)*: block B1 is on top of block B2
- *ontable(B)*: block B is on the table
- *clear(B)*: block B has no blocks on top of it and is not being held by the gripper



Action	Preconditions	Add List	Delete List
unstack(B1, B2)	empty & clear(B1) & on(B1, B2)	holding(B1), clear(B2)	empty, on(B1, B2), clear(B1)
pickup(B)	empty & clear(B) & ontable(B)	holding(B)	empty, ontable(B), clear(B)
stack(B1, B2)	holding(B1) & clear(B2)	empty, on(B1, B2), clear(B1)	clear(B2), holding(B1)
putdown(B)	holding(B)	empty, ontable(B), clear(B)	holding(B)

PDDL: Planning Domain Definition Language

PDDL (Planning Domain Definition Language) is a standard encoding language for “classical” planning tasks

- *Objects*: Things in the world that interest us
- *Predicates*: Properties of objects that we are interested in (*true/false)*.
- *Initial state*: The state of the world that we start in.
- *Goal specification*: Things that we want to be true.
- *Actions/Operators*: Ways of changing the state of the world.

Planning tasks specified in PDDL are separated into two files

- A *domain file* for predicates and actions
- A *problem file* for objects, initial state and goal specification

PDDL was invented in 1998 for the first IPC and nowadays most common planners read PDDL files …

PDDL: Domain files

(define (domain <DOMAIN_NAME>)
    (:requirements :strips)
    (:predicates (<PREDICATE_1_NAME> ?<arg_1> ?<arg_2> ...)
        (<PREDICATE_2_NAME> ...)
        ...)
    (:action <ACTION_1_NAME>
        :parameters (?<par_1> ?<par_2> ...)
        :precondition <*COND_FORMULA: PartiallySpecifiedState*>
        :effect <*EFFECT_FORMULA: ADDlist + DELETElist*>
)
    (:action <ACTION_2_NAME>
        ...)
    ...)

PDDL: Problem Files

(define (problem <PROBLEM_NAME>)
    (:domain <DOMAIN_NAME>)
    (:objects <obj_1> <obj_2> ...)
    (:init <$ATOM_1$> <$ATOM_2$> ...)
    (:goal <*COND_FORMULA:* *PartiallySpecifiedState*>)
)

Where we have:

- Init and Goal are *ground*! (not parameterised, i.e., not ?x kind of things)
- *COND_FORMULA:* conjunction of atoms
  (AND $atom_1$ ... $atom_n$)
- *EFFECT_FORMULA:* conjunction of ADDED & DELETED (NOT) atoms
  (AND $atom_1$ ... (NOT $atom_n$))

Basic PDDL Example: Gripper Domain

Gripper task with four balls:

There is a robot that can move between two rooms and pick up or drop balls with either of his two arms. Initially, all balls and the robot are in the first room. We want the balls to be in the second room.

- Objects: The two rooms, four balls and two robot arms.
- Predicates: Is x a room? Is x a ball? Is ball x in room y? Is robot arm x empty? [...]
- Initial state: All balls and the robot are in the first room. All robot arms are empty. [...]
- Goal specification All balls must be in the second room.
- Actions/Operators: The robot moves between rooms, pick up a ball or drop a ball.

Gripper Domain: Objects

Objects in the gripper domain

- Rooms: rooma, roomb
- Balls: ball1, ball2, ball3, ball4
- Robot arms: left, right

In PDDL without typing

- (:objects rooma roomb ball1 ball2 ball3 ball4 left right)

In PDDL with typing

- (:types room ball robot-arm)
- (:objects rooma – room roomb – room
  ball1 – ball ball2 – ball ball3 – ball ball4 – ball
  left – robot-arm right – robot-arm)

Gripper Domain: Predicates (without typing)

Predicates in the gripper domain without typing

- ROOM(x) – true iff x is a room
- BALL(x) – true iff x is a ball
- GRIPPER(x) – true iff x is a gripper (robot arm)
- at-robby(x) – true iff x is a room and the robot is in x
- at-ball(x, y) – true iff x is a ball, y is a room, and x is in y
- free(x) – true iff x is a gripper and x does not hold a ball
- carry(x, y) – true iff x is a gripper, y is a ball, and x holds y

In PDDL this translates into:

- (:predicates
  (ROOM ?x) (BALL ?x) (GRIPPER ?x)
  (at-robby ?x) (at-ball ?x ?y)
  (free ?x) (carry ?x ?y)
)

Gripper Domain: Predicates (with typing)

Predicates in the gripper domain with typing

- at-robby(x) – true iff x is a room and the robot is in x
- at-ball(x, y) – true iff x is a ball, y is a room, and x is in y
- free(x) – true iff x is a gripper and x does not hold a ball
- carry(x, y) – true iff x is a gripper, y is a ball, and x holds y

In PDDL this translates into:

- (:predicates
  (at-robby ?x – room)
  (at-ball ?x – balll ?y – room)
  (free ?x – robot-arm)
  (carry ?x – robot-arm ?y – ball)
)

Gripper Domain: Initial State

The Initial state (according to the example text):

- ROOM(rooma) and ROOM(roomb) are true.
- BALL(ball1), ..., BALL(ball4) are true.
- GRIPPER(left), GRIPPER(right), free(left) and free(right) are true.
- at-robby(rooma), at-ball(ball1, rooma), ..., at-ball(ball4, rooma) are true.
- Everything else is false.

In PDDL this translate into:

- (:init
  (ROOM rooma) (ROOM roomb)
  (BALL ball1) (BALL ball2) (BALL ball3) (BALL ball4)
  (GRIPPER left) (GRIPPER right) (free left) (free right)
  (at-robby rooma) (at-ball ball1 rooma) (at-ball ball2 rooma)
  (at-ball ball3 rooma) (at-ball ball4 rooma)
)

Gripper Domain: Goal State

The Goal state (according to the example text):

- at-ball(ball1, roomb), ..., at-ball(ball4, roomb) must be true.
- Everything else we don't care about.

In PDDL this translates into:

- (:goal
  (and (at-ball ball1 roomb)
  (at-ball ball2 roomb)
  (at-ball ball3 roomb)
  (at-ball ball4 roomb)
)
)

Gripper Domain: Movement Operator

The robot can move from x to y:

- Precondition: ROOM(x), ROOM(y) and at-robby(x) are true.
- Effect: at-robby(y) becomes true and at-robby(x) becomes false.
- Everything else doesn't change.

In PDDL this translates into:

- (:action move
  :parameters (?x ?y)
  :precondition (and (ROOM ?x) (ROOM ?y) (at-robby ?x))
  :effect (and (at-robby ?y) (not (at-robby ?x)))
)

Gripper Domain: Pick-up Operator

The robot can pick up x in y with z.

- Precondition: BALL(x), ROOM(y), GRIPPER(z), at-ball(x, y), at-robby(y) and free(z) are true.
- Effect: carry(z, x) becomes true while at-ball(x, y) and free(z) become false.
- Everything else doesn't change.

In PDDL this translates into:

- (:action pick-up
  :parameters (?x ?y ?z)
  :precondition (and (BALL ?x) (ROOM ?y) (GRIPPER ?z)
  (at-ball ?x ?y) (at-robby ?y) (free ?z))
  :effect (and (carry ?z ?x) (not (at-ball ?x ?y)) (not (free ?z)))
)

Gripper Domain: Drop Operator

The robot can drop x in y from z

- Precondition: BALL(x), ROOM(y), GRIPPER(z), carry(z,x), at-robby(y) are true.
- Effect: at-ball(x, y) and free(z) become true while carry(z, x) becomes false.
- Everything else doesn't change.

In PDDL this translates into:

- (:action drop :parameters (?x ?y ?z)
  :precondition (and (BALL ?x) (ROOM ?y) (GRIPPER ?z)
  (carry ?z ?x) (at-robby ?y))
  :effect (and (at-ball ?x ?y) (free ?z) (not (carry ?z ?x)))
)

Let's solve it!

Using satplan to solve the gripper problem

- Download satplan (2006 version, winner of IPC)
  - http://www.cs.rochester.edu/users/faculty/kautz/satplan/index.htm
  - tar -zxvf SatPlan2006.tgz

- Compile satplan by issuing
  - cd SatPlan2006
  - make

- Run vanilla satplan (i.e., default options)
  - cd include/bin/
  - ./satplan -path ../../gripper/ -domain gripper_domain.pddl -problem gripper_problem.pddl

- Observe the plan
  - less gripper_problem.pddl.soln

PDDL 1.2 (IPC 2000)

In successive revisions of the language requirements where added:

- :strips
- :typing *in :predicates, :parameters and :objects*
- :equality **=**
- :negativepreconditions **not**
- :disjunctivepreconditions **or**
- :existentialpreconditions **exists**
- :universalpreconditions **forall**
- :quantifiedpreconditions = :existentialpreconditions + :universalpreconditions
- :conditionaleffects **when**
- :adl *= all the above (Action Description Language)*

PDDL: Typing in Domain and Problem Files

(define (domain <DOMAIN_NAME>)
  (:requirements :strips :typing)
  (:types <$type_1$> <$type_2$> ...)
  (:predicates (<PREDICATE_1_NAME> ?<arg_1> - <$type_1$> ...)
    (<PREDICATE_2_NAME> ...))
  (:action <ACTION_1_NAME>
    :parameters (?<par_1> - <$type_1$> ?<par_2> - <$type_2$> ...)
    :precondition < *COND_FORMULA:* *PartiallySpecifiedState*>
    :effect < *EFFECT_FORMULA:* *ADDlist + DELETElist*>)
…)

(define (problem <PROBLEM_NAME>)
  (:domain <DOMAIN_NAME>)
  (:objects <obj_1> - <$type_1$> <obj_2> - <$type_2$> ...)
  (:init <$ATOM_1$> <$ATOM_2$> ...)
  (:goal < *COND_FORMULA:* *PartiallySpecifiedState* >)
)

STRIPS vs ADL Conditional Formulas

The *:requirement* clause defines the power of the language that should be understood by the planner

- :strips
  - Conjunction of atoms — (AND $atom_1$ ... $atom_n$)
  - If :equality added atoms my be in the form — (= arg_1 arg_2)
  - Only positive

- :adl
  - equality (=) — (= arg_1 arg_2)
  - negation (NOT) — (NOT $atom_1$)
  - conjunction (AND) — (AND $atom_1$ ... $atom_n$)
  - disjunction (OR) — (OR $atom_1$ ... $atom_n$)
  - quantifier — (FORALL, EXISTS)

(FORALL (?v - t) (*PREDICATE* ?v))

(EXISTS (?v - t) (*PREDICATE* ?v))

STRIPS vs ADL Effect Formulas

The *:requirement* clause defines the power of the language that should be understood by the planner

- :strips
  - Conjunction of added and deleted atoms (AND $atom_1$ ... (NOT $atom_n$))
- :adl
  - Conditional effect:

    (WHEN *PRECOND_FORMULA* EFFECT_FORMULA)
  - Universal quantified formula:

    (FORALL (?<v_1> - <t_1> ?<v_2> - <t_2>) *EFFECT_FORMULA*)

PDDL 2.1: Time (the idea)

A feasible plan is sometimes not enough, thus a new version of planner was introduced to take into account:

- Durative actions: time 
- Fluents: numbers
- Metrics: optimal plan

Time in planning (scheduling)

- actions take time to execute
- how long an action takes to execute may depend on the preconditions
- preconditions may need to hold when the action begins, or throughout its execution
- effects may not be true immediately and they may persist for only a limited time
- an action can have multiple effects on a fluent at different times

PDDL 2.1: Time (the code)

A feasible plan is sometimes not enough, thus a new version of planner was introduced to take into account:

- Durative actions: time
- Fluents: numbers
- Metrics: optimal plan

In the Domain file

- (:durative-action <name>
  :parametes (…)
  :duration (= ?duration <time>)
  :condition (…)
  :effect (…))
- CONDITIONAL_FORMULA: at_start, overall, at_end
- EFFECT_FORMULA: at_start, at_end

PDDL 2.1: Resources (the idea)

A feasible plan is sometimes not enough, thus a new version of planner was introduced to take into account:

- Durative actions: time
- Fluents: numbers
- Metrics: optimal plan

Resources in planning

- A resource is any quantity or (set of) object(s) whose value or availability determines whether an action can be executed
- Resources may be consumable (examples: money, fuel) or reusable (example: a car which becomes available again after a trip)
- In some cases, actions may produce resources (examples: refueling, hiring more staff, etc)
- When planning with resources, a solution is defined as a plan that achieves the goals while allocating resources to actions so that all resource constraints are satisfied

PDDL 2.1: Resources (the code)

A feasible plan is sometimes not enough, thus a new version of planner was introduced to take into account:

- Durative actions: time
- Fluents: numbers
- Metrics: optimal plan

In the Domain definition

- (:functions (<name1> ?<obj1> - <type1>)
  (<name2> ?<obj2> - <type2>)
  (…))
- CONDITIONAL FORMULA: = > < <= => + - * /
- EFFECT FORMULA:
  - assign, increase, decrease, scale-up, scale-down

In the Problem definition

- (:init (= (<ATOM>) <#>))

PDDL 2.1: Metrics (the idea)

A feasible plan is sometimes not enough, thus a new version of planner was introduced to take into account:

- Durative actions: time
- Fluents: numbers
- Metrics: optimal plan ⟸

Optimal planning (and scheduling)

- As with search problems, we can distinguish between optimal and satisficing solutions
- A satisficing plan is one that achieves the goal(s) without violating any temporal or resource constraints
- An optimal plan is one that achieves the goal(s) while minimising (or maximising) some metric (metric is often defined in terms of resource usage)

PDDL 2.1: Metrics (the code)

A feasible plan is sometimes not enough, thus a new version of planner was introduced to take into account:

- Durative actions: time
- Fluents: numbers
- Metrics: optimal plan 

In the problem definition

- (:metric minimize[maximize] <objective_function>)

Built-in function:

- total-time

PDDL 2.1 – IPC-2002

“We have a Four Gallon Jug of Water and a Three Gallon Jug of Water and a Water Pump.
The challenge of the problem is to be able to put exactly two gallons of water in the Four Gallon Jug, even though there are no markings on the Jugs.”

Drew McDermott.
“The 1998 AI Planning Systems Competition”.
AI Magazine (21):2, 2000.