Planning

Preceded by Logic Agents

CIS 32

Functionalia

HW 4 Part A :

DO the Neural Network Problem from HW 2. No one did it and it will be on the Final.

Today:

Constructing the Logical Agent

Planning

STRIPS Planning

HW 3

Logic-Based Agents

- When we started talking about logic, it was as a means of representing knowledge.
- We wanted to represent knowledge in order to be able to build agents.
- We now know enough about logic to do that.
- We will now see how a *logic-based agent* can be designed to perform simple tasks.
- Assume each agent has a *database*, i.e., set of FOL-formulae.

  These represent information the agent has about environment

Notation

- We'll write Δ for this database.
- Also assume agent has set of *rules* (called ***R***).
- We write $\Delta \vdash_R \phi$ if the formula ϕ can be proved from the database Δ using only the rules ***R***.

- How to program an agent:

  Write the agent's rules ***R*** so that it should do action ***a***

  whenever $\Delta \vdash_R Do(a)$.

Here, *Do* is a predicate.

- Also assume ***A*** is set of actions agent can perform.

Logic Based Agent Algorithm

The agent's operations is as followed:

1. for each a in A do
2. if $\Delta \vdash_R Do(a)$ then
3. return a
4. end-if
5. end-for
6. for each a in A do
7. if $\Delta \nvdash_R \neg Do(a)$ then
8. return a
9. end-if
10. end-for
11. return $null$

Example :Vacuum Robot

We have a small robot that will clean up a house.

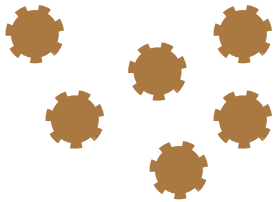
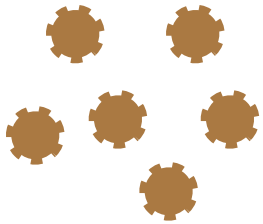
The robot has a **sensor** to tell it whether it is over any dirt, and a **vacuum** that can be used to suck up dirt. Robot always has an orientation (one of *n*, *s*, e, or *w*). Robot can **move** forward one "step" or **turn** right 90 degrees.

The agent moves around a room, which is divided grid-like into a number of equally sized squares. Assume that the room is a 3 by 3 grid, and agent starts in square (0, 0) facing north.

What are some domain predicates that we can use to describe the world and robot state?

Next Slide Shows Picture

Robot Environment

 *dirt* (0,2)	 *dirt* (1,2)	(2,2)
(0,1)	 (1,1)	(2,1)
(0,0)	(1,0)	(2,0)

- Three domain predicates in this exercise:

In(x, y)	agent is at *(x, y)*
Dirt(x, y)	there is dirt at *(x, y)*
Facing(d)	the agent is facing direction *d*

- For convenience, we write rules as (form of *Horn Clause*):

$$\phi(\ldots) \longrightarrow \psi(\ldots)$$

- Three domain predicates in this exercise:

In(x, y)	agent is at *(x, y)*
Dirt(x, y)	there is dirt at *(x, y)*
Facing(d)	the agent is facing direction *d*

- For convenience, we write rules as (form of *Horn Clause*):

$$\phi(\ldots) \longrightarrow \psi(\ldots)$$

- First rule deals with the basic cleaning action of the agent

$$In(x, y) \wedge Dirt(x, y) \longrightarrow Do(suck)$$

- *Hardwire the basic navigation algorithm*, so that the robot will always move from (0, 0) to (0, 1) to (0, 2) then to (1, 2), (1, 1) and so on.

- Once agent reaches (2, 2), it must head back to (0, 0).

$$In(0,0) \wedge Facing(north) \wedge \neg Dirt(0,0) \longrightarrow Do(forward) \quad (2)$$
$$In(0,1) \wedge Facing(north) \wedge \neg Dirt(0,1) \longrightarrow Do(forward) \quad (3)$$
$$In(0,2) \wedge Facing(north) \wedge \neg Dirt(0,2) \longrightarrow Do(turn) \quad (4)$$
$$In(0,2) \wedge Facing(east) \longrightarrow Do(forward) \quad (5)$$

- Other considerations:
  - adding new information after each move/action;
  - removing old information.
- Suppose we scale up to 10 × 10 grid?

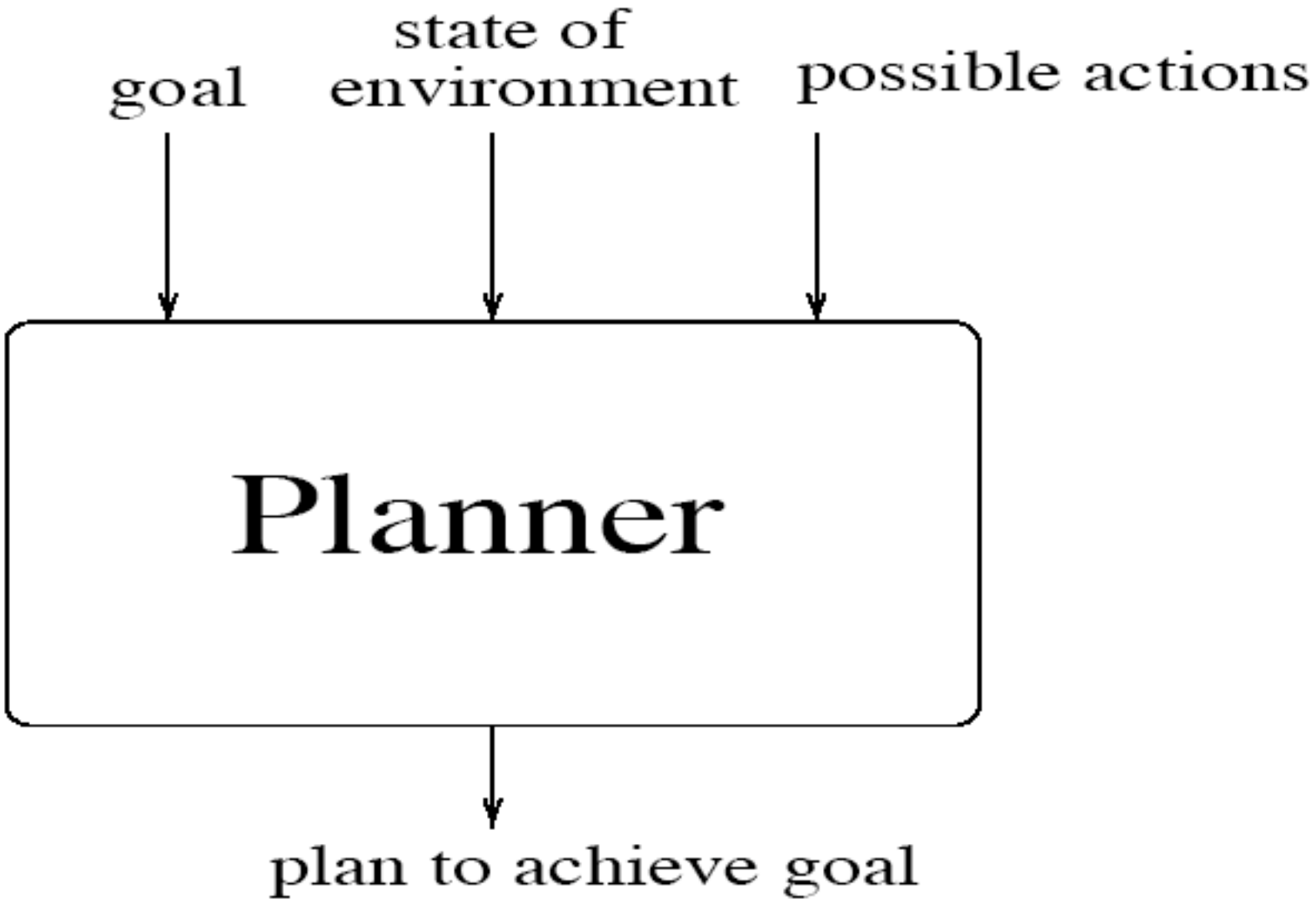
What is planning?

- Key problem facing *agent is deciding what to do.*
- We want agents to be *taskable*: give them *goals* to achieve, have them decide for themselves how to achieve them.
- Basic idea is to give an agent:
  - representation of goal to achieve;
  - knowledge about what actions it can perform; and
  - knowledge about state of the world;

  and to have it generate a *plan* to achieve the goal.
- Essentially, this is

automatic programming

High Level Box View of a Planner



- **Question**: How do we *represent*. . .
  - goal to be achieved;
  - state of environment;
  - actions available to agent;
  - plan itself.

Language *expressive* enough to describe a wide variety of problems.

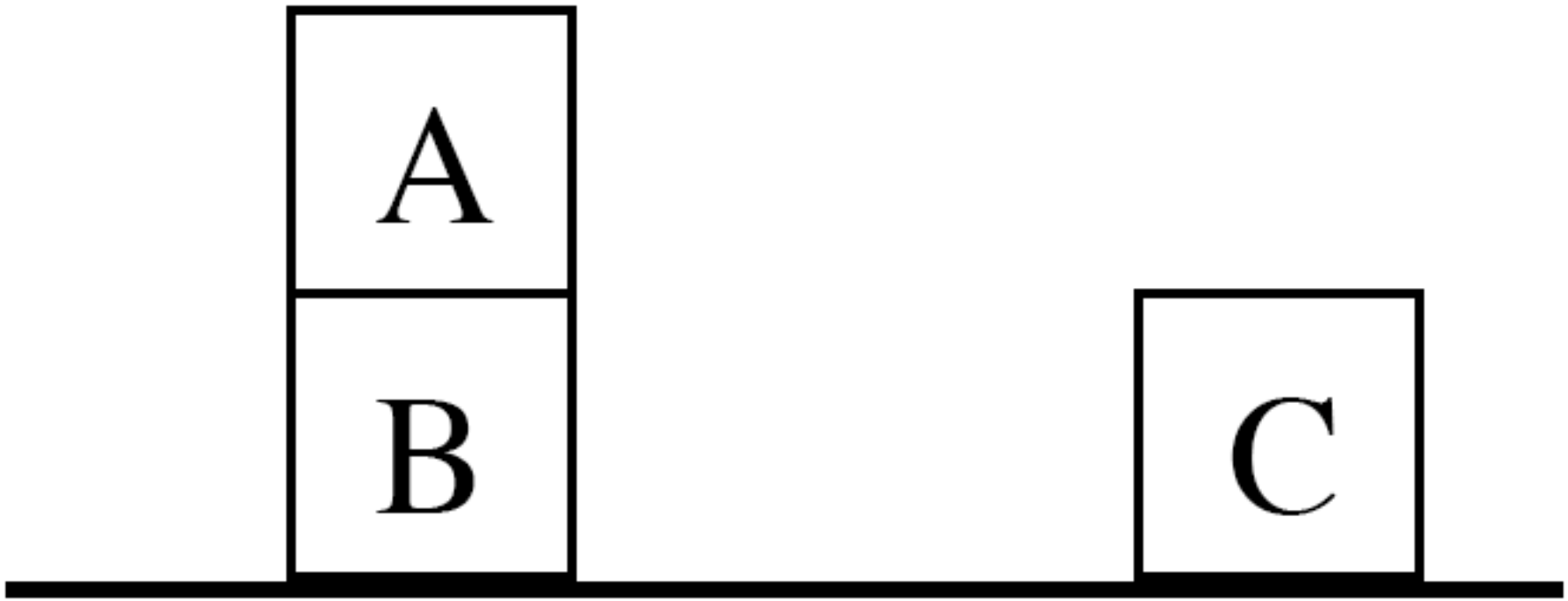
Language *restrictive* enough to allow efficient operation over them.

STRIPS Language (using First Order Logic).

STandford **R**esearch **I**nstitute **P**roblem **S**olver.

Blocks World

- We'll illustrate the techniques with reference to the *blocks world*.
- Contains a robot arm, 3 blocks (A, B and C) of equal size, and a table-top.
- Initial state:



- To represent this environment, need an *ontology*

On(x, y)	obj *x* on top of obj *y*
OnTable(x)	obj *x* is on the table
Clear(x)	nothing is on top of obj *x*
Holding(x)	arm is holding *x*

• Here is a first-order logic representation of the blocks world described above:

Clear(A)
On(A,B)
OnTable(B)
OnTable(C)
Clear(C)

• Uses *ground* literals (function-free)

not Clear(x,y), or Clear(OnTopOf(B))

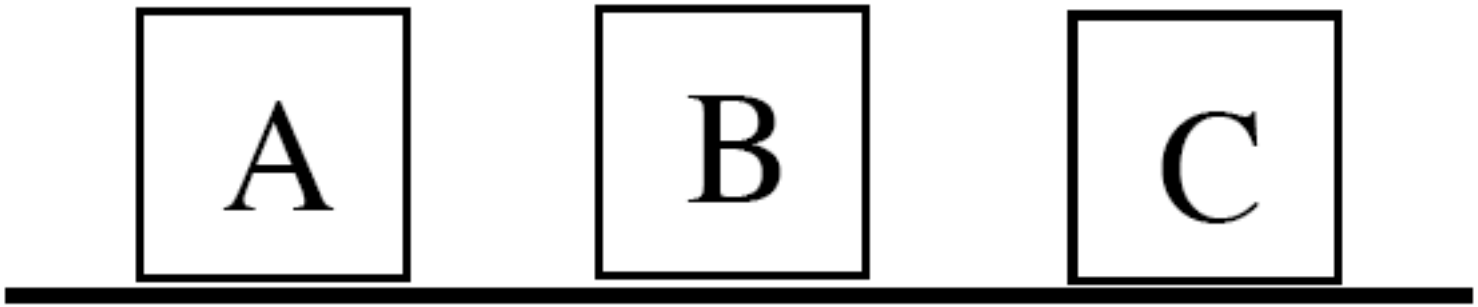
• Use the *closed world assumption*: anything not stated is assumed to be *false*

• A *goal* is represented as a first-order logic formula. (Conjunction of positive ground literals)

• Here is a goal:

$$OnTable(A) \wedge OnTable(B) \wedge OnTable(C)$$

• Which corresponds to the state:



• *Actions* are represented using a technique that was developed in the STRIPS planner.

- Each action has:
  - a *name*
    - which may have arguments;
  - a *pre-condition list*
    - list of facts which must be true for action to be executed;
  - a *delete list*
    - list of facts that are no longer true after action is performed;
  - an *add list*
    - list of facts made true by executing the action.

Each of these may contain variables

Stack

- **Example 1**:

The *stack* action occurs when the robot arm places the object *x* it is holding is placed on top of object *y*.

	Stack(x, y)
pre	*Clear(y)* $\wedge$ *Holding(x)*
del	*Clear(y)* $\wedge$ *Holding(x)*
add	*ArmEmpty* $\wedge$ *On(x, y)*

Unstack

- **Example 2**:

The *unstack* action occurs when the robot arm picks an object *x* up from on top of another object *y*.

	UnStack(x, y)
pre	$On(x, y) \wedge Clear(x) \wedge ArmEmpty$
del	$On(x, y) \wedge ArmEmpty$
add	$Holding(x) \wedge Clear(y)$

Stack and UnStack are *inverses* of one-another.

Pickup

- **Example 3**:

The *pickup* action occurs when the arm picks up an object *x* from the table.

	Pickup(x)
pre	$Clear(x) \wedge OnTable(x) \wedge ArmEmpty$
del	$OnTable(x) \wedge ArmEmpty$
add	*Holding(x)*

Putdown

- **Example 4**:

The *putdown* action occurs when the arm places the object *x* onto the table.

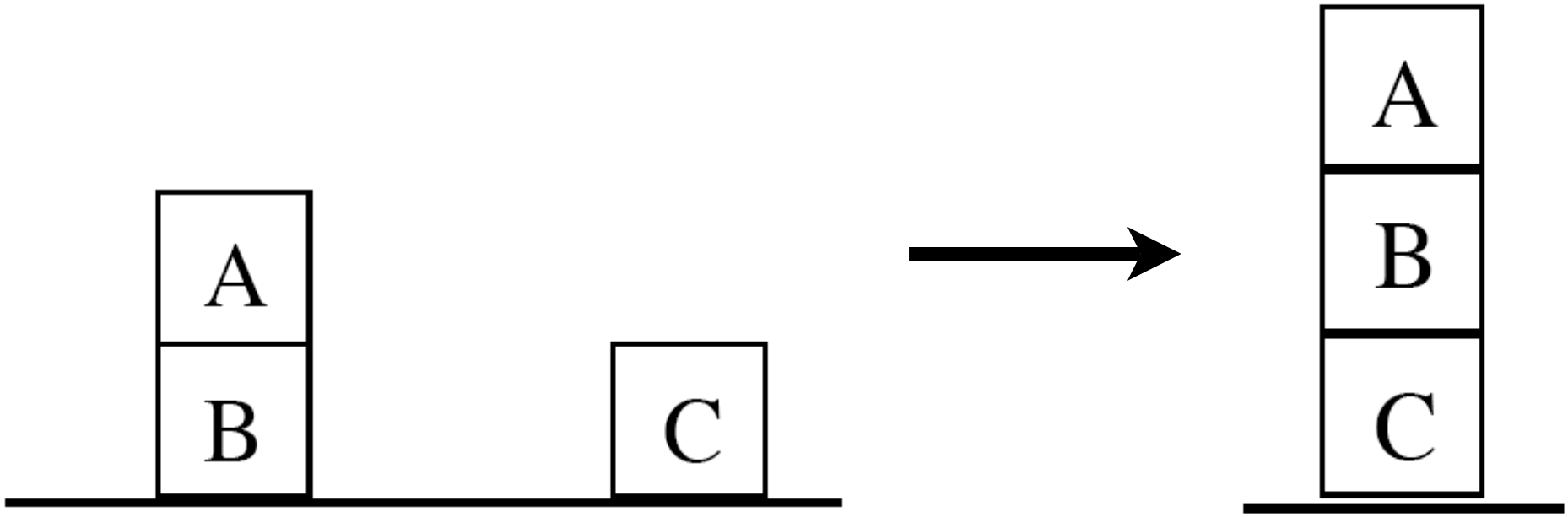
	PutDown(x)
pre	*Holding(x)*
del	*Holding(x)*
add	*OnTable(x)* $\wedge$ *ArmEmpty*

- What is a plan?

  A sequence (list) of actions, with variables replaced by constants.

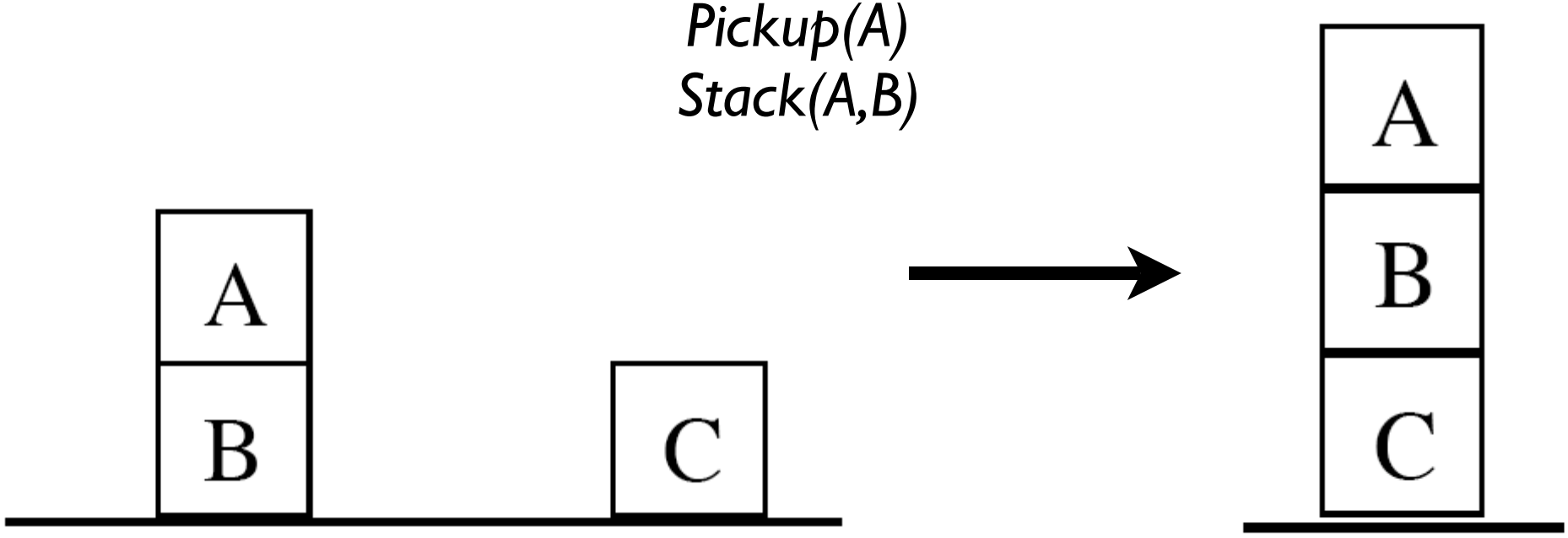
  Constants being: ***A, B, C, Floor***

- So, to get from:



- We need the set of actions:

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



- In "*real life*", plans contain conditionals (IF ..THEN...) and loops (WHILE... DO...), but most simple planners cannot handle such constructs — they construct *linear plans*.

- Simplest approach to planning: *means-ends analysis*.

- Involves backward chaining from **goal** to original **start state**.

  1. Start by finding an action that has goal as post-condition. (Assume this is the *last* action in plan.)

  2. Then figure out what the previous state would have been.

  3. Try to find action that has *this* state as post-condition.

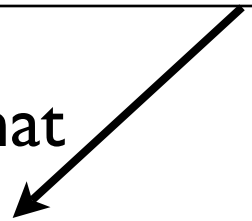
- *Recurse* until we end up (hopefully!) in original state

function *plan*(

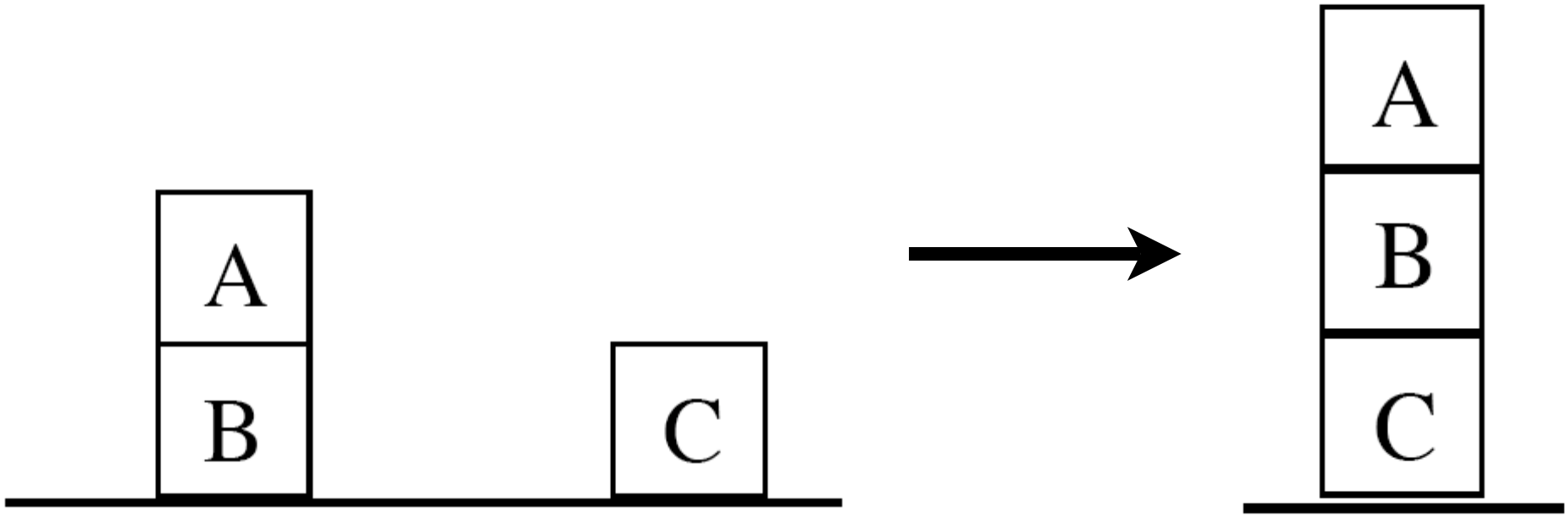
d : WorldDesc, // initial env state
g : Goal, // goal to be achieved
p : Plan, // plan so far
A : set of actions // actions available)

1. if $d \models g$ then
2. return *p*
3. else
4. choose *a* in *A* such that
5. $add(a) \models g$ and
6. $del(a) \not\models g$
7. set *g* = *pre(a)*
8. append *a* to *p*
9. return *plan(d, g, p,A)*

All positive effects of ***a*** that appear in ***g*** are deleted.
Each precondition literal of ***a*** is added, unless it already appears.



How does this work on the previous example?



How does this work on the previous example?

Start

OnTable(C)
OnTable(B)
On(A,B)
Clear(A)
Clear(C)
Clear(Floor)
ArmEmpty

Goal

OnTable(C)
On(B,C)
On(A,B)
Clear(A)
Clear(Floor)
ArmEmpty

	Stack(x, y)
pre	*Clear(y) ∧ Holding(x)*
del	*Clear(y) ∧ Holding(x)*
add	*ArmEmpty ∧ On(x, y)*

	UnStack(x, y)
pre	*On(x, y) ∧ Clear(x) ∧ ArmEmpty*
del	*On(x, y) ∧ ArmEmpty*
add	*Holding(x) ∧ Clear(y)*

	Pickup(x)
pre	*Clear(x) ∧ OnTable(x) ∧ ArmEmpty*
del	*OnTable(x) ∧ ArmEmpty*
add	*Holding(x)*

	PutDown(x)
pre	*Holding(x)*
del	*Holding(x)*
add	*OnTable(x) ∧ ArmEmpty*

Start

OnTable(C)
OnTable(B)
On(A,B)
Clear(A)
Clear(C)
Clear(Floor)
ArmEmpty

Goal

OnTable(C)
On(B,C)
On(A,B)
Clear(A)
Clear(Floor)
ArmEmpty

	Stack(x, y)
pre	Clear(y) ∧ Holding(x)
del	Clear(y) ∧ Holding(x)
add	ArmEmpty ∧ On(x, y)

	UnStack(x, y)
pre	On(x, y) ∧ Clear(x) ∧ ArmEmpty
del	On(x, y) ∧ ArmEmpty
add	Holding(x) ∧ Clear(y)

	Pickup(x)
pre	Clear(x) ∧ OnTable(x) ∧ ArmEmpty
del	OnTable(x) ∧ ArmEmpty
add	Holding(x)

	PutDown(x)
pre	Holding(x)
del	Holding(x)
add	OnTable(x) ∧ ArmEmpty

Start

OnTable(C)
OnTable(B)
On(A,B)
Clear(A)
Clear(C)
Clear(Floor)
ArmEmpty

Goal

OnTable(C)
On(B,C)
On(A,B)
Clear(A)
Clear(Floor)
ArmEmpty

- This algorithm not guaranteed to find the plan. . .

. . . but it is *sound*: If it finds the plan that is correct.

- Some problems:
  - negative goals;
  - maintenance of goals;
  - conditionals & loops;
  - exponential search space;

The Frame Problem

- A general problem with representing properties of actions:

  How do we know exactly what changes as the result of performing an action?

  If I pick up a block, does my hair color stay the same?

- One solution is to write *frame axioms*.

Here is a frame axiom, which states that *CHIPP*'s hair color is the same in all the situations *(symbolized by s and s')* that result from performing *Pickup(x)* in situation *s* as it is in *s'*.

$$\forall s, s'.\ Result(CHIPP, Pickup(x), s) = s' \Rightarrow$$

$$HairColor(CHIPP, s) = HairColor(CHIPP, s')$$

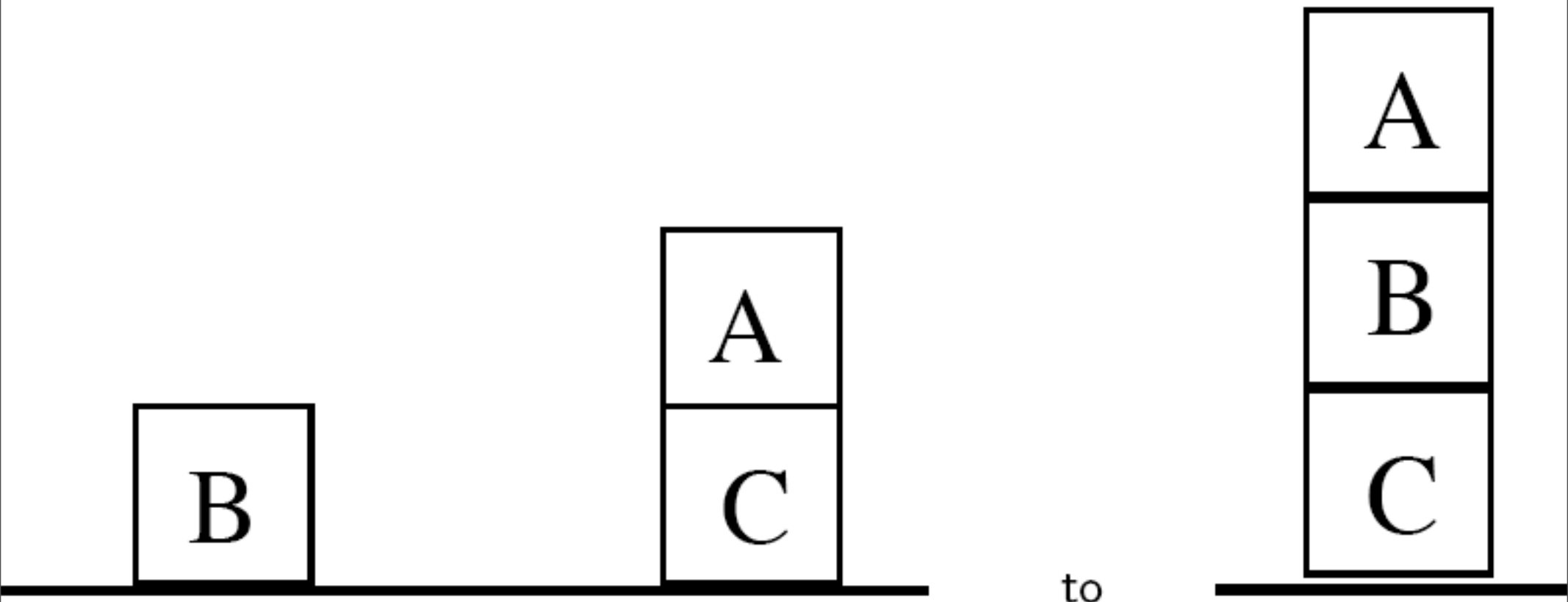
STRIPS Planning

- Stating frame axioms in this way is unfeasible for real problems.
- (Think of all the things that we would have to state in order to cover all the possible frame axioms).
- STRIPS solves this problem by assuming that everything not explicitly stated to have changed remains unchanged.
- The price we pay for this is that we lose the advantages of using logic:
  - Semantics goes out of the window
- However, more recent work has effectively solved the frame problem (using clever second-order approaches).

Second-order Logic (BTW) - involved quantification across different sets

Sussman's Anomaly

- Consider we have the following initial state and goal state:



- What operations will be in the plan?

• Clearly we need to *Stack* B on C at some point, and we also need to *Unstack* A from C and *Stack* it on B.

• Which operation goes first?

• Obviously we need to do the *UnStack* first, and the *Stack B* on *C*, but the planner has no way of knowing this.

• It also has no way of "undoing" a partial plan if it leads into a dead end.

• So if it chooses to *Stack(A,C)* after the *Unstack*, it is sunk.

• This is a big problem with linear planners

• How could we modify our planning algorithm?

• Modify the middle of the algorithm to be:

1. if $d \models g$ then
2. return p
3. else
4. choose a in A such that
5. $add(a) \models g$ and
6. $del(a) \not\models g$

6a. *no clobber(add(a), del(a), rest of plan)*

7. set $g = pre(a)$
8. append a to p
9. return *plan(d, g, p,A)*

• We do this with *partial-order planning*.