

# Planning system

- One of the important key issue of *AI-planning*
- *Planning is the process of computing steps of problem solving before executing them*
- It increases the autonomy and flexibility in the system
- It combines two major areas –logic and search
- Planner may be a
  - Program that searches for a solution
  - Program that proves the existence of the solution
- It integrates partial solutions to generate a final solution---decomposable problems
- It also solves non decomposable problems

# Components of planning system

- 1. Chose the best rule**
- 2. Apply the chosen rule**
- 3. Detect when a solution has been found.**
- 4. Detecting dead ends**
- 5. Repairing an Almost Correct Solution**

# Components of a Planning System

## 1. **Chose the best rule** to be applied(based on heuristics).

- The most widely used technique for selecting an appropriate rules to apply is
  - isolate a set of differences between desired goal state and current state.
  - identify those rules that are relevant to reducing those differences.
  - If several rules, use heuristic information to choose among them

# Components of a Planning System

- 2. Apply the chosen rule** to compute the new problem state that arises from its application.
  - In simple systems, applying rules is easy.
  - Each rule simply describe the problem state that would result from its application.
  - In complex systems, we must be able to deal with rules that specify only a small part of the complete problem state.
  - One way is to describe, for each action, each of the changes it makes to the state description

# Components of a Planning System

## **3. Detect when a solution has been found.**

- How will it know when this has been done?
- In simple problem-solving systems, this question is easily answered by a straightforward match of the state descriptions.
- By the time required goal state is reached, solution of each sub goal is found
- Solutions so obtained are then combined to form the final solution of the problem

# Components of a Planning System

## **4. Detecting dead ends**

- As a planning system is searching for a sequence of operators to solve a particular problem, it must be able to detect when it is exploring a path that can never lead to a solution-----dead ends

# Components of a Planning System

## **5. Repairing an Almost Correct Solution**

- The kinds of techniques we are discussing are often useful in solving nearly decomposable problems.
- One good way of solving such problems is to assume that they are completely decomposable, proceed to solve the sub problems separately, and then check that when the sub solutions are combined, they do in fact yield a solution to the original problem.

# Types of planning

- Linear planning
  - One goal is solved at a time
  - Simple search strategy—stack of unachieved goals
  - Advantages
    - Since one goal is solved at a time search space is reduced
    - It is helpful only when goals are independent
  - Disadvantage
    - It may not produce an optimal goal
    - May be incomplete at times
    - Planning efficiency depends on ordering of goal



# Types of planning

- Non-linear planning
  - Sub goals may be solved in any order
  - Sub goals may be interdependent
  - It uses a goal set instead of goal stack
  - All possible sub goals ordering are considered and goal interaction are handled
  - Adv
    - It is sound and complete
    - May produce an optimal goal
  - Disadv
    - As all sub goal ordering are considered large search space is required
    - More complex algorithm

# Block World Problem: Description

- It consists of handling of blocks and generating new pattern from given pattern.  
Assumptions are:
  - All blocks are of same size—square in shape
  - Blocks can be stacked on each other
  - There is a flat surface (table) on which blocks are placed
  - There is robot arm that can manipulate the blocks
  - Robot arm can hold only one block at a time

# State description

- In Block world problem, states are described by the set of predicates
- Predicate represent the fact that are true in that state
- When an action/operator is applied we describe the change that action make to state description
- when action is applied
  - Some facts change where as some facts remain unchanged
  - Example, when block is picked up, its color do not change

# The Blocks World

- Operators:
  - **UNSTACK(A,B)** – Pick up block A from its current position on block B. The arm must be empty and block A must have no blocks on top of it.
  - **STACK(A,B)** – Place block A on block B. The arm must already be holding A and the surface of B must be clear.

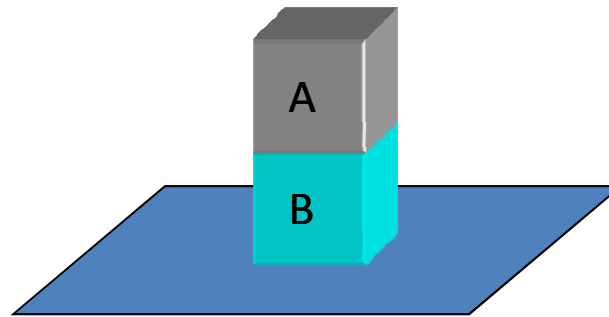
# The Blocks World cont.

- **PICKUP(A)** – Pick up block A from the table and hold it. The arm must be empty and there must be nothing on top of block A.
- **PUTDOWN(A)** – Put block A down on the table. The arm must have been holding block A.

# The Blocks World cont.

- Predicates:
  - **ON(A,B)** – Block A is on block B.
  - **ONTABLE(A)** – Block A is on the table.
  - **CLEAR(A)** – There is nothing on top of block A.
  - **HOLDING(A)** – The arm is holding block A.
  - **ARMEMPTY** – The arm is holding nothing.

# A Simple Blocks World Description



ON(A,B,S0)  $\wedge$   
ONTABLE(B,S0)  $\wedge$   
CLEAR(A,S0)

# STRIPS – Style Operate for the Blocks World

## STACK(x,y)

P:  $\text{CLEAR}(y) \wedge \text{HOLDING}(x)$

D:  $\text{CLEAR}(y) \wedge \text{HOLDING}(x)$

A:  $\text{ARMEMPTY} \wedge \text{ON}(x,y)$

## UNSTACK(x,y)

P:  $\text{ON}(x,y) \wedge \text{CLEAR}(x) \wedge \text{ARMEMPTY}$

D:  $\text{ON}(x,y) \wedge \text{ARMEMPTY}$

A:  $\text{HOLDING}(x) \wedge \text{CLEAR}(y)$



# STRIPS – Style Operate for the Blocks World cont.

## PICKUP(x)

P:  $\text{CLEAR}(x) \wedge \text{ONTABLE}(x) \wedge \text{ARMEMPTY}$

D:  $\text{ONTABLE}(x) \wedge \text{ARMEMPTY}$

A:  $\text{HOLDING}(x)$

## PUTDOWN(x)

P:  $\text{HOLDING}(x)$

D:  $\text{HOLDING}(x)$

A:  $\text{ONTABLE}(x) \wedge \text{ARMEMPTY}$

# State description

- For simple problem it is easy to record and maintain the fact that do not change when a operator is applied
- $S_0$ —intial state
- $S_1$  is state after applying operator
- $S_1$  must include all facts that are changed and those that are not changed
- Such system—frame axioms
- Size of such system gets larger in complex domain
- So better mechanism is one which do not require large frame axioms to stated explicitly

# STRIPS

- STandford Research Institute Problem Solver
- Developed by Fikes and Nilson in 1971
- Each operator is described by the
  - A list of new predicate that become true---**ADD** list
  - A list of predicate that become false----**DEL** list
  - A list of predicate that must be true before a operator is applied...**PRE** (Precondition list)

# STRIPS representation

- Plan operators

*STACK(x,y), UNSTACK(x,y), PICKUP(x), PUTDOWN(x)*

- Predicates:

*ON(x,y), ONTABLE(x), CLEAR(x), HOLD(x), ARMEMPTY*

- Axioms:

$$(\exists x) (\text{HOLD}(x)) \rightarrow \sim \text{ARMEMPTY}$$

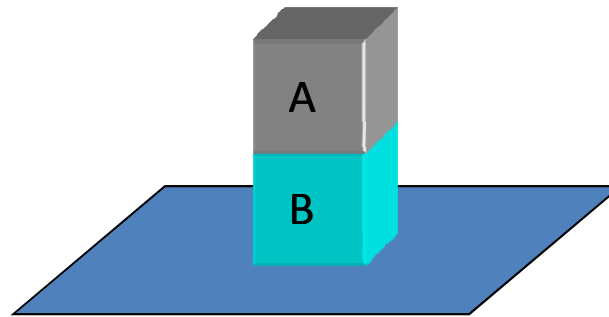
$$(\forall x) (\text{ONTABLE}(x) \rightarrow \sim (\exists y) (\text{ON}(x,y)))$$

$$(\forall x) (\sim (\exists y) (\text{ON}(y,x)) \rightarrow \text{CLEAR}(x))$$

# STRIPS style

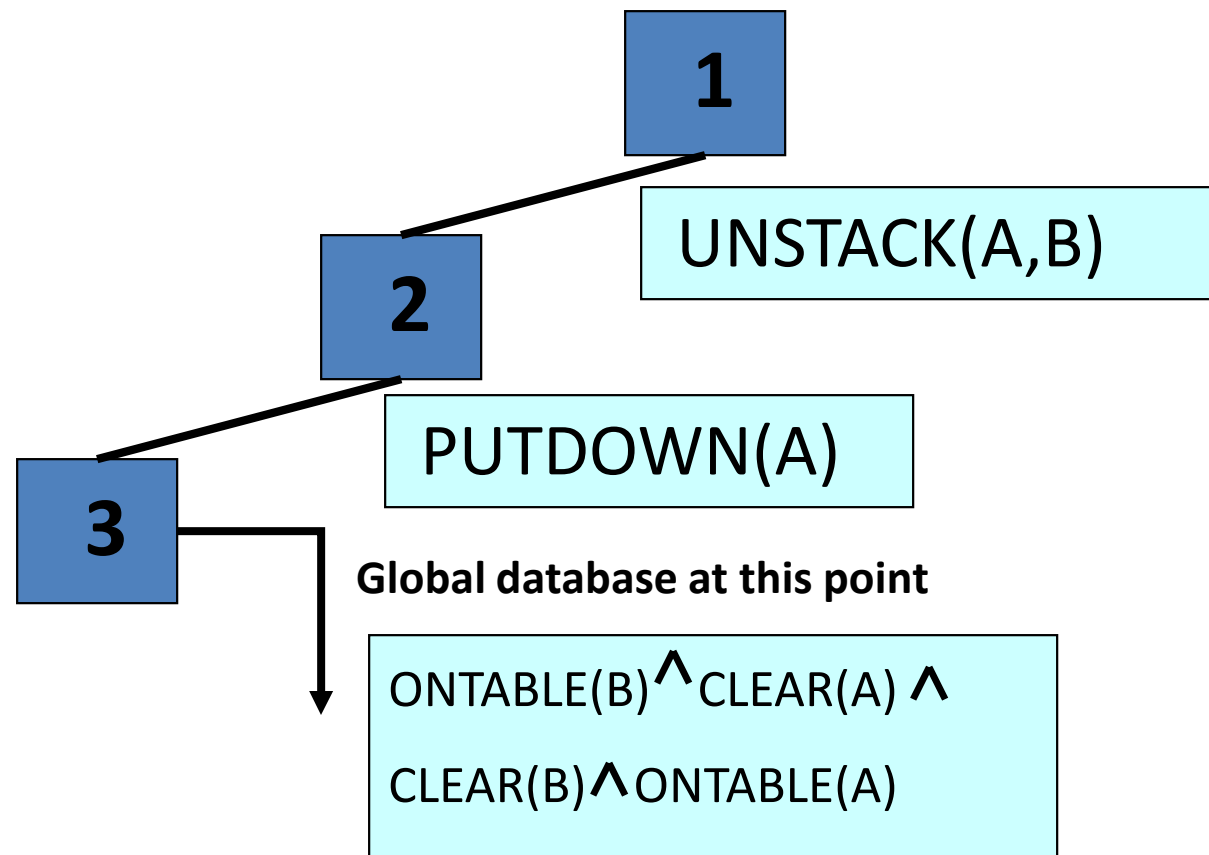
- **STACK(x,y)**
  - P: CLEAR(y)^HOLDING(x)
  - D: CLEAR(y)^HOLDING(x)
  - A: ARMEMPTY^ON(x,y)
- **PICKUP(x)**
  - P: CLEAR(x) ^ ONTABLE(x)  
^ARMEMPTY
  - D: ONTABLE(x) ^  
ARMEMPTY
  - A: HOLDING(x)
- **UNSTACK(x,y)**
  - P: ON(X,Y) ^CLEAR(X)^ARMEMPTY
  - D:ON(X,Y) ^CLEAR(X)^ARMEMPTY
  - A: HOLDING(X)^CLEAR(Y)
- **PUTDOWN(x)**
  - P: HOLDING(X)
  - D: HOLDING(X)
  - A: ARMEMPTY ^ONT(X)

# A Simple Blocks World Description

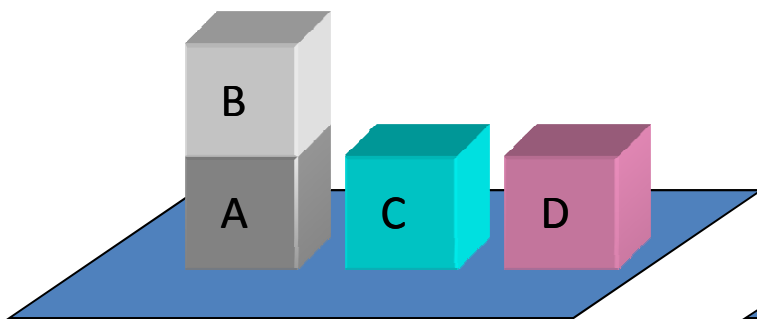


ON(A,B,S0)     $\wedge$   
ONTABLE(B,S0)     $\wedge$   
CLEAR(A,S0)

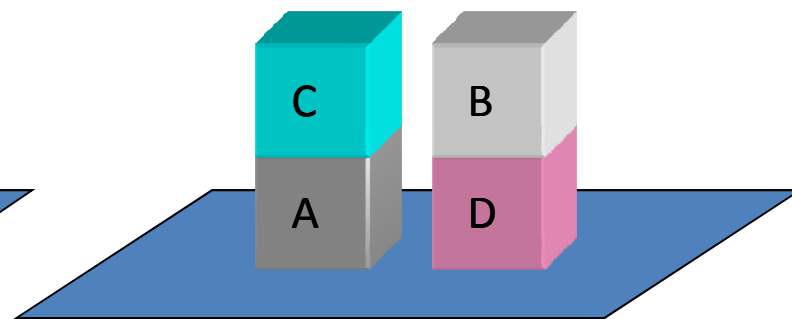
# A Simple Search Tree



# A Very Simple Blocks World Problem



Start:  $\text{ON}(\text{B}, \text{A}) \wedge$   
 $\text{ONTABLE}(\text{A}) \wedge$   
 $\text{ONTABLE}(\text{C}) \wedge$   
 $\text{ONTABLE}(\text{D}) \wedge$   
 $\text{ARMEMPTY}$



Goal:  $\text{ON}(\text{C}, \text{A}) \wedge$   
 $\text{ON}(\text{B}, \text{D}) \wedge$   
 $\text{ONTABLE}(\text{A}) \wedge$   
 $\text{ONTABLE}(\text{D})$



# Goal Stack Planning

## 1) Initial goal stack:

$ON(C,A) \wedge ON(B,D) \wedge ONTABLE(A) \wedge ONTABLE(D)$

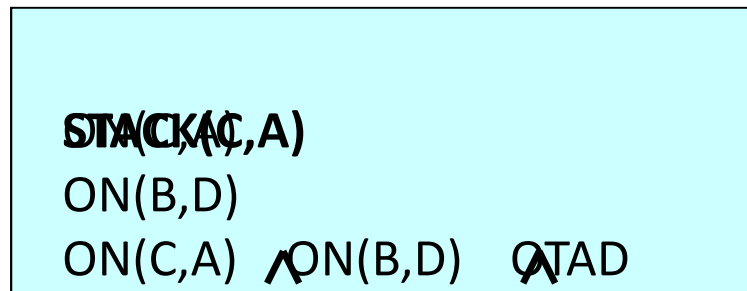
# Goal Stack Planning cont.

2) Choose to work on  $ON(C,A)$  before  $ON(B,D)$ :

$ON(C,A)$   
 $ON(B,D)$   
 $ON(C,A) \wedge ON(B,D) \wedge OTAD$

# Goal Stack Planning cont.

3) Achieve  $ON(C,A)$  with  $STACK(C,A)$ :



# Goal Stack Planning cont.

4) Add STACK's preconditions:

CLEAR(A)  
HOLDING(C)  
CLEAR(A)  $\wedge$  HOLDING(C)  
**STACK(C,A)**  
ON(B,D)  
ON(C,A)  $\wedge$  ON(B,D)  $\wedge$  OTAD

# Goal Stack Planning cont.

5) Achieve CLEAR(A) with UNSTACK(B,A):

```

ON(B,A)
CLEAR(B)
ARMEMPTY
ON(B,A) ^ CLEAR(B) ^ ARMEMPTY
ONSTACK(B,A)
HOLDING(C)
CLEAR(A) ^ HOLDING(C)
STACK(C,A)
ON(B,D)
ON(C,A) ^ ON(B,D) ^ ATAD
    
```

# Goal Stack Planning cont.

## 6) Pop satisfied predicates:

ON(B,A)  
CLEAR(B)  
ARMEMPTY  
~~ON(B,A)~~ ~~CLEAR(B)~~ ~~ARMEMPTY~~  
**UNSTACK(B,A)**  
HOLDING(C)  
~~CLEAR(A)~~ ~~HOLDING(C)~~  
**STACK(C,A)**  
ON(B,D)  
~~ON(C,A)~~ ~~ON(B,D)~~ ~~ATAD~~

# Goal Stack Planning cont.

7) Achieve HOLDING(C) with UNSTACK(C,x):

```

ON(C,x)
CLEAR(C)
ARMEMPTY
ON(C,x) ^ CLEAR(C) ^ ARMEMPTY
UNSTACK(C,x)
CLEAR(A) ^ HOLDING(C)
STACK(C,A)
ON(B,D)
ON(C,A) ^ ON(B,D) ^ ATAD
    
```

# Goal Stack Planning cont.

8) Achieve ON(C,x) by STACK(C,x):

```

CLEAR(x)
HOLDING(C)
CLEAR(x) HOLDING(C)
STACK(C,x)
CLEAR(C)
ARMEMPTY
ON(C,x) CLEAR(C) ARMEMPTY
UNSTACK(C,x)
CLEAR(A) HOLDING(C)
STACK(C,A)
ON(B,D)
ON(C,A) ON(B,D) OTAD
    
```



# Goal Stack Planning cont.

9) Terminate path because HOLDING(C) is duplicated.

```
CLEAR(x)
HOLDING(C)
CLEAR(x) HOLDING(C)
STACK(C,x)
CLEAR(C)
ARMEMPTY
ON(C,x) ^ CLEAR(C) ^ ARMEMPTY
UNSTACK(C,x)
CLEAR(A) HOLDING(C)
STACK(C,A)
ON(B,D)
ON(C,A) ^ ON(B,D) ^ QAD
```

# Goal Stack Planning cont.

10) Achieve HOLDING(C) with PICKUP, not UNSTACK:

```
ONTABLE(C)
CLEAR(C)
ARMEMPTY
  ONTABLE(C)  ^ CLEAR(C)  ^ ARMEMPTY
PICKUP(C)
  CLEAR(A)  ^ HOLDING(C)
STACK(C,A)
  ON(B,D)
  ON(C,A)  ^ ON(B,D)  ^ OTAD
```

# Goal Stack Planning cont.

11) Pop ONTABLE(C) and CLEAR(C), and achieve ARMEMPTY by STACK(B,D):

```

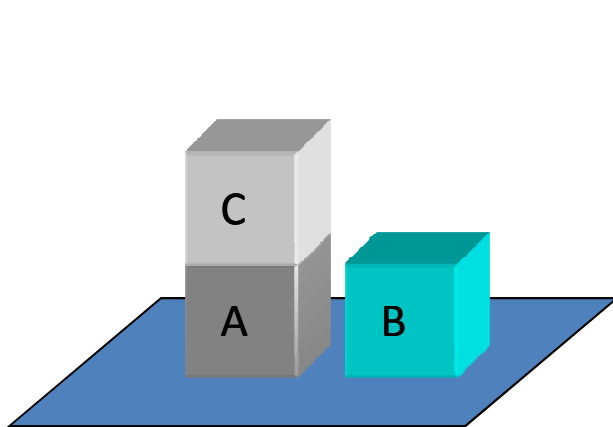
CLEAR(D)
ONTABLE(B)
CLEAR(D) HOLDING(B)
STACK(B,D)
ONTABLE(C) CLEAR(C) ARMEMPTY
PICKUP(C)
CLEAR(A) HOLDING(C)
STACK(C,A)
ON(B,D)
ON(C,A) ON(B,D) OTAD
    
```

# Goal Stack Planning cont.

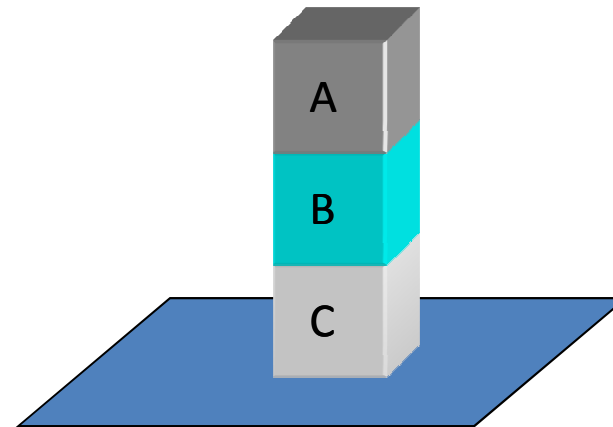
12) Pop entire stack, and return plan:

- i. UNSTACK(B,A).
- ii. STACK(B,D).
- iii. PICKUP(C).
- iv. STACK(C,A).

# A Slightly Harder Blocks Problem



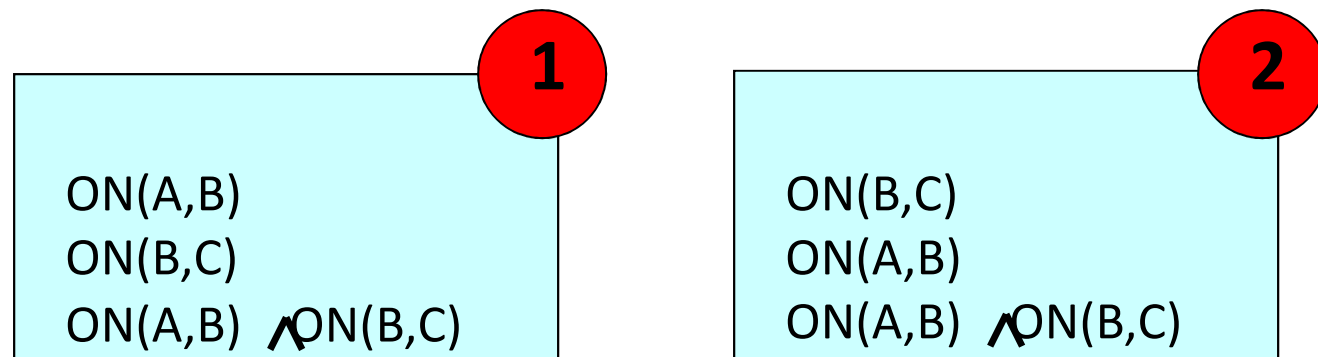
Start:  $\text{ON}(\text{C}, \text{A}) \wedge$   
 $\text{ONTABLE}(\text{A}) \wedge$   
 $\text{ONTABLE}(\text{B}) \wedge$   
 $\text{ARMEMPTY}$



Goal:  $\text{ON}(\text{A}, \text{B}) \wedge$   
 $\text{ON}(\text{B}, \text{C})$

# Goal Stack Planning

- There are two ways to begin solving:



# Goal Stack Planning cont.

- Let's choose alternative 1. we will eventually produce this goal stack:

```

CLEAR(C)
ARMEMPTY
CLEAR(C) ARMEMPTY
ONSTACK(C,A)
ARMEMPTY
CLEAR(A) ARMEMPTY
PICKUP(A)
CLEAR(B) HOLDING(A)
STACK(B,B)
ON(B,C)
ON(A,B) ON(B,C)
    
```

# Goal Stack Planning cont.

- We can pop off the stack goals that have already been satisfied:

To satisfy ARMEMPTY we need to PUTDOWN(C).

```

CLEAR(C)
ARMEMPTY
CLEAR(C)  ARMEMPTY
HOLDING(C,A)
PUTDOWN(C)
CLEAR(A)  ARMEMPTY
PICKUP(A)
CLEAR(B)  HOLDING(A)
STACK(A,B)
ON(B,C)
ON(A,B)  ON(B,C)
    
```



# Goal Stack Planning cont.

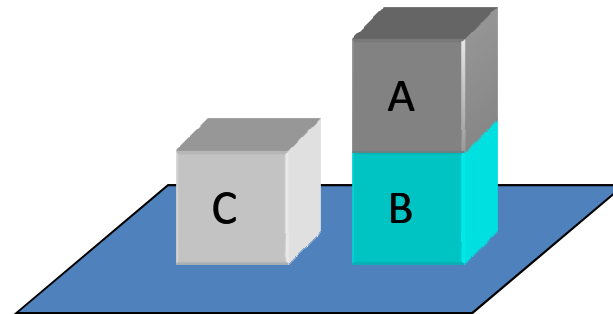
- We can continue popping:

```
HOLDING(C)
PUTDOWN(C)
CLEAR(A) ARMEMPTY
PICKUP(A)
CLEAR(B) HOLDING(A)
STACK(A,B)
ON(B,C)
ON(A,B) ON(B,C)
```

# Goal Stack Planning cont.

- The current state is:

ONTABLE(B)	^
ON(A,B)	^
ONTABLE(C)	^
ARMEMPTY	



- The sequence of operators applied so far is:
  1. UNSTACK(C,A)
  2. PUTDOWN(C)
  3. PICKUP(A)
  4. STACK(A,B)

# Goal Stack Planning cont.

- Now we can begin to work on  $ON(B,C)$ .

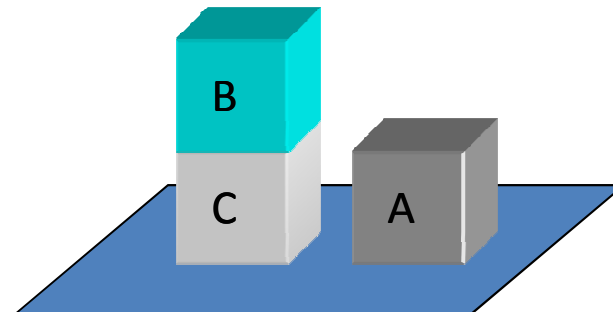
We need to stack B on C. To do that we need to unstack A from B. By the time we have achieved the goal  $ON(B,C)$ , and popped it off the stack, we will have executed the following additional sequence of operators



# Goal Stack Planning cont.


- The sequence of operators :
  5. UNSTACK(A,B)
  6. PUTDOWN(A)
  7. PICKUP(B)
  8. STACK(B,C)
- The problem state will be:

ON(B,C)	^
ONTABLE(A)	^
ONTABLE(C)	^
ARMEMPTY	



# Goal Stack Planning cont.

- But now when we check the remaining goal on the stack:



ON(A,B) ^ ON(B,C)

we discover that it is not satisfied.

We have undone ON(A,B) in the process of achieving ON(B,C).

- The sequence of operators we need to add:
  9. PICKUP(A)
  10. STACK(A,B)

# Goal Stack Planning cont.

The complete plan that has been discovered is:

1. UNSTACK(C,A)
2. PUTDOWN(C)
3. PICKUP(A)
4. STACK(A,B)
5. UNSTACK(A,B)
6. PUTDOWN(A)
7. PICKUP(B)
8. STACK(B,C)
9. PICKUP(A)
10. STACK(A,B)

❖ Although this plan will achieve the desired goal, it does not do so very efficiently.

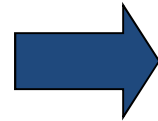
❖ The same would have happened if we had chosen the second alternative.

# Goal Stack Planning cont.

- There are two approaches we can take to the question of how a good plan could be found:
  1. Look at ways to repair the plan we already have – look for places in the plan where we perform an operation and then immediately undo it. We can eliminate both the doing and the undoing steps from the plan.
  2. Use a plan-finding procedure that could construct efficient plans directly – the Nonlinear Planning.

# Repairing the First Plan

1. UNSTACK(C,A)
2. PUTDOWN(C)
3. PICKUP(A)
4. STACK(A,B)
5. UNSTACK(A,B)
6. PUTDOWN(A)
7. PICKUP(B)
8. STACK(B,C)
9. PICKUP(A)
10. STACK(A,B)

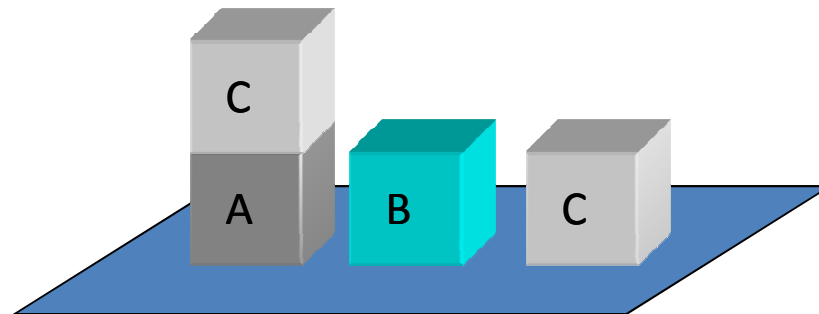


1. UNSTACK(C,A)
2. PUTDOWN(C)
3. PICKUP(B)
4. STACK(B,C)
5. PICKUP(A)
6. STACK(A,B)



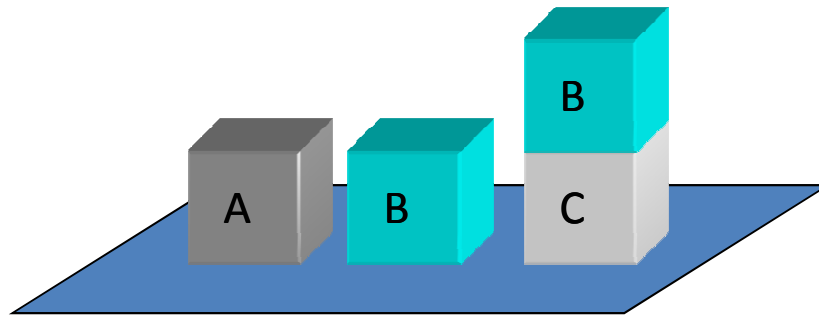
# The Nonlinear Planning

- Begin work on the goal  $ON(A,B)$  by clearing A, thus putting C on the table.



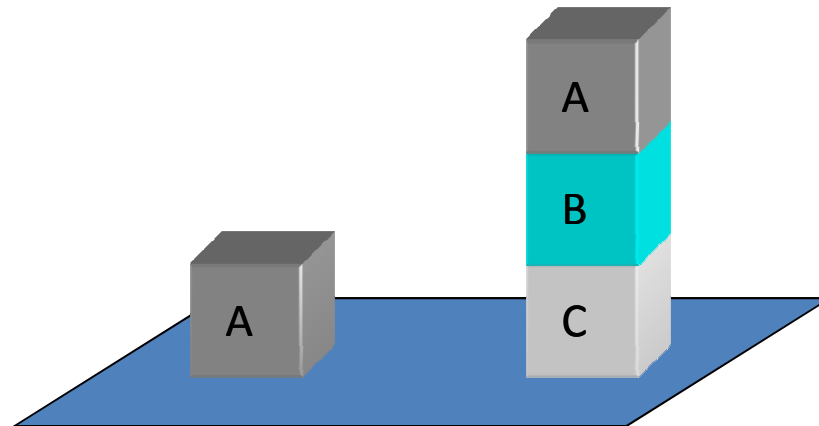
# Nonlinear Planning cont.

- Achieve the goal  $ON(B,C)$  by stacking B on C.



# Nonlinear Planning cont.

- Complete the goal  $ON(A,B)$  by stacking A on B.



# Backwards Non-Linear Planner

- Finds “Last Operator” recursively
- Uses theorem prover to prune paths

# Non Linear Planning

# Partial Ordering

- Any planning algorithm
  - that can place two actions into a plan without specifying which comes first is called a *partial-order planner*.
  - actions dependent on each other are ordered in relation to themselves but not necessarily in relation to other independent actions.
- The solution is represented as a *graph* of actions, not a sequence of actions.

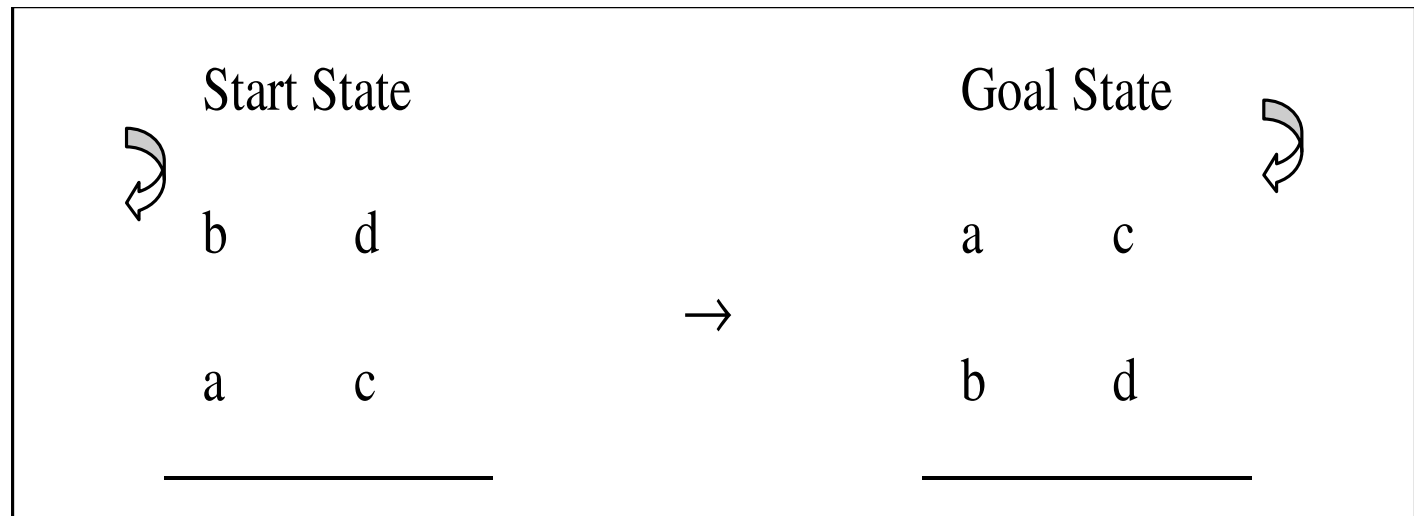
## ***Cont..***

- Let us define following two macros for the sake of simplicity for block world example.

Macro Operator	Description	Body
MOT(X)	Move X onto table	US(X, _), PD(X)
MOVE(X, Y)	Move X onto Y	PU(X), ST(X, Y)

## Cont...

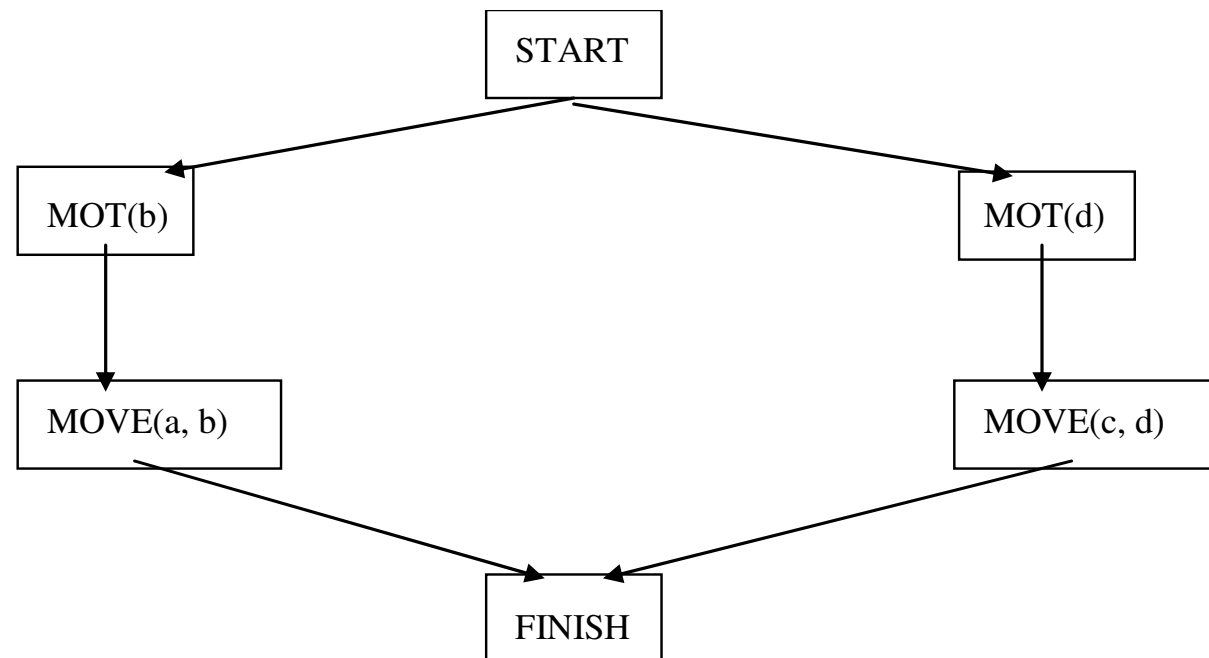
- To achieve Goal state ON(a,b),
  - move 'b' onto table should occur before move 'a' to 'b'
  - Hence partial ordering  $MOT(b) \leftarrow MOVE(a, b)$  holds true
  - $MOT(b)$  should come before  $MOVE(a, b)$  in the final plan.
- Similarly to achieve Goal state ON(c,d),
  - partial order  $MOT(d) \leftarrow MOVE(c, d)$  is established.





# ***Partial Graph***

- Partial graph contains the dummy actions START and FINISH to mark the beginning and end of the plan in the graph.
- The planner can generate total plans from the graph.



# ***Total Plans Generation***

- From this representation total six plans are generated.
- Each of these is called a *linearization* of the partial-order plan.

Different Total Plans					
Plan1	Plan2	Plan3	Plan4	Plan5	Plan6
MOT(b) MOVE(a, b) MOT(d) MOVE(c, d)	MOT(b) MOT(d) MOVE(a, b) MOVE(c, d)	MOT(b) MOT(d) MOVE(c, d) MOVE(a, b)	MOT(d) MOVE(c, d) MOT(b) MOVE(a, b)	MOT(d) MOT(b) MOVE(c, d) MOVE(a, b)	MOT(d) MOT(b) MOVE(a, b) MOVE(c, d)

# Nonlinear Planning - Constraint Posting

- Idea of constraint posting is to build up a plan by incrementally
  - hypothesizing operators,
  - partial ordering between operators and
  - binding of variables within operators
- At any given time in planning process, a solution is a partially ordered.
- To generate actual plan, convert the partial order into total orders.

# ***Steps in Non Linear Plan Generation***

- Step addition
  - Creating new operator (step) for a plan
- Promotion
  - Constraining one operator to come before another in final plan
- Declobbering
  - Placing operator Op2 between two operators Op1 and Op3 such that Op2 reasserts some pre conditions of Op3 that was negated by Op1
- Simple Establishment
  - Assigning a value to a variable, in order to ensure the pre conditions of some step.

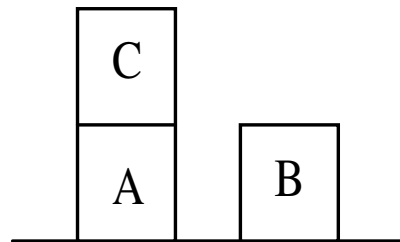
# *Algorithm*

1. Initialize S to be set of propositions in the goal state.
2. Remove some unachieved proposition P from S.
3. Achieve P by using step addition, promotion, declobbering, simple establishment.
4. Review all the steps in the plan, including any new steps introduced by step addition to see if any of their preconditions are unachieved.
5. Add to S the new set of unachieved preconditions.
6. If  $S = \emptyset$ , complete the plan by converting the partial order of steps into a total order and instantiate any variables as necessary and exit.
7. Otherwise go to step 2.

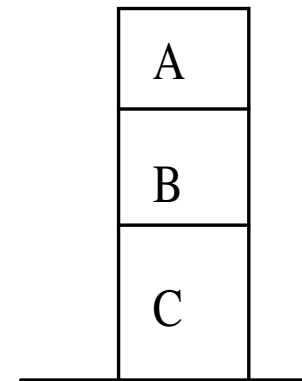
## ***Example: Sussman anomaly problem***

- Begin with null plan (no operators).
- Look at the goal state and find the operators that can achieve them.
- There are two operators (steps) Stack(A, B) and Stack(B,C) which have post conditions as ON(A,B) and ON(B, C).

Initial State (State0)



Goal State



Initial State:  $ON(C, A) \wedge ONT(A) \wedge ONT(B) \wedge AE \wedge CL(C) \wedge CL(B)$

Goal State:  **$ON(A, B) \wedge ON(B, C)$**

## Cont...

Pre Cond	Clear(B) *Holding(A)	Clear(C ) *Holding(B)
Operator	Stack(A, B)	Stack(B,C)
Post Cond	ON(A, B) AE ~ Clear(B) ~ Holding(A)	ON(B,C) AE ~ Clear(C ) ~ Holding(B)

- Here unachieved conditions are marked with \*.
- HOLD in both the cases is not true as Armempty is true initially.
- Introduce new operator (step) to achieve these goals.
- This is called operator (step) **addition**.
- Add PickUp operator on both the goals.

# Cont...

Pre Con	*Clear(A) ONT(A) *AE	*Clear(B ) ONT(B) *AE
Operator	<b>Pickup(A)</b>	<b>Pickup(B)</b>
Post Cond	Holding(A) ~ ONT(A) ~ AE ~ Clear(A)	Holding(B) ~ ONT(B) ~ AE ~ Clear(B )
Pre Con	Clear(B) *Holding(A)	Clear(C ) *Holding(B)
Operator	<b>Stack(A, B)</b> ON(A, B)	<b>Stack(B,C)</b> ON(B,C)
Post Cond	AE ~ Clear(B) ~ Holding(A)	AE ~ Clear(C ) ~ Holding(B)



## Cont..

- It is clear that in a final plan, Pickup must precede STACK operator.
- Introduce the ordering as follows:
  - Whenever we employ operator, we need to introduce ordering constraints called **promotion**.

---

Plan 1

---

Pickup(A)  $\leftarrow$  Stack(A, B)

Pickup(B)  $\leftarrow$  Stack(B, C)

---

- Here we partially ordered operators and four unachieved pre conditions:- Clear(A), Clear(B ), Armempty on both the paths
  - Clear(A) is unachieved as C is on A in initial state.
  - Also Clear(B) is unachieved even though top of B is clear in initial state but there exist a operator Stack(A,B) with post condition as  $\sim$ Clear(B).

Initial State: **ON(C, A)**  $\wedge$  ONT(A)  $\wedge$  ONT(B)  $\wedge$  Armempty  $\wedge$  Clear(C)  $\wedge$  Clear(B)

## Cont..

- If we make sure that Pickup(B) precede Stack(A, B) then **Clear(B)** is achieved. So post the following constraints.

---

### Plan 1

Pickup(A)  $\leftarrow$  Stack(A, B)

Pickup(B)  $\leftarrow$  Stack(B, C)

---

### Plan2

Pickup(B)  $\leftarrow$  Stack(A, B)

---

- Note that pre cond Clear(A) of Pickup(A) still is unachieved.
  - Let us achieve Armempty preconditions of each Pick up operators before Clear(A).
  - Initial state has AE. So one PU can achieve its pre cond but other PU operator could be prevented from being executed.
  - Assume Armempty is achieved as pre condition of Pickup(B) as its other preconditions have been achieved. So put constraint.

## Cont..

- Similarly, following plans are generated

Plan3

Pickup(B)  $\leftarrow$  Pickup(A) (pre conds of Pickup(A) are not still achieved.)

- Since Pickup(B) makes  $\sim$ Armempty and Stack(B,C) will make Armempty which is precondition of Pickup(A), we can put the following constraint.

Plan4

Pickup(B)  $\leftarrow$  **Stack(B, C)**  $\leftarrow$  Pickup(A)

- Here Pickup(B) is said to clobber pre condition of Pickup(A) and Stack(B, C) is said to declobber it. (removing deadlock)

## ***Cont..***

---

### Plan 5

---

Unstack(C, A)  $\leftarrow$  Stack(B, C)

Unstack(C, A)  $\leftarrow$  Pickup(A)

Unstack(C, A)  $\leftarrow$  Pickup(B)

---

- Declobbering:

---

### Plan 6

---

Unstack(C, A)  $\leftarrow$  PutDown(C)  $\leftarrow$  Pickup(B)

---

## Cont..

- Combine the following partial plans to generate final plan.

---

Pickup(A)  $\leftarrow$  Stack(A, B)

Pickup(B)  $\leftarrow$  Stack(B, C)

Pickup(B)  $\leftarrow$  Stack(A, B)

Pickup(B)  $\leftarrow$  Pickup(A)

(pre conds of Pickup(A) are not still achieved.)

---

Pickup(B)  $\leftarrow$  Stack(B, C)  $\leftarrow$  Pickup(A)

---

Unstack(C, A)  $\leftarrow$  Stack(B, C)

Unstack(C, A)  $\leftarrow$  Pickup(A)

Unstack(C, A)  $\leftarrow$  Pickup(B)

Unstack(C, A)  $\leftarrow$  PutDown(C)  $\leftarrow$  Pickup(B)

---

Final plan: Unstack(C,A)  $\leftarrow$  PutDown(C)  $\leftarrow$  Pickup(B)  $\leftarrow$  Stack(B,C)  $\leftarrow$  Pickup(A)  $\leftarrow$  Stack(A,B)

# Learning Plans

- In many problems, plans may share a large number of common sequence of actions.
- So planner requires the ability to recall and modify or reuse the existing plans.
- **Macro operators** can be defined as sequence of operators for performing some task and saved for future use.
- Example:
  - **Reverse\_blocks(X, Y)** can be macro operator with plan – Unstack(X, Y), PutDown(X), Pickup(Y), Stack(Y, X), where X, Y are variables.
- The generalized plan schema is called MACROP and is stored in a data structure called **Triangle table**.
- It helps planner to build new plans efficiently by using existing plans.