Artificial Intelligence:
Search Methods for Problem Solving

Planning

A First Course in Artificial Intelligence: Chapter 7 & 10

Deepak Khemani
Department of Computer Science & Engineering
IIT Madras

An action view of problem solving

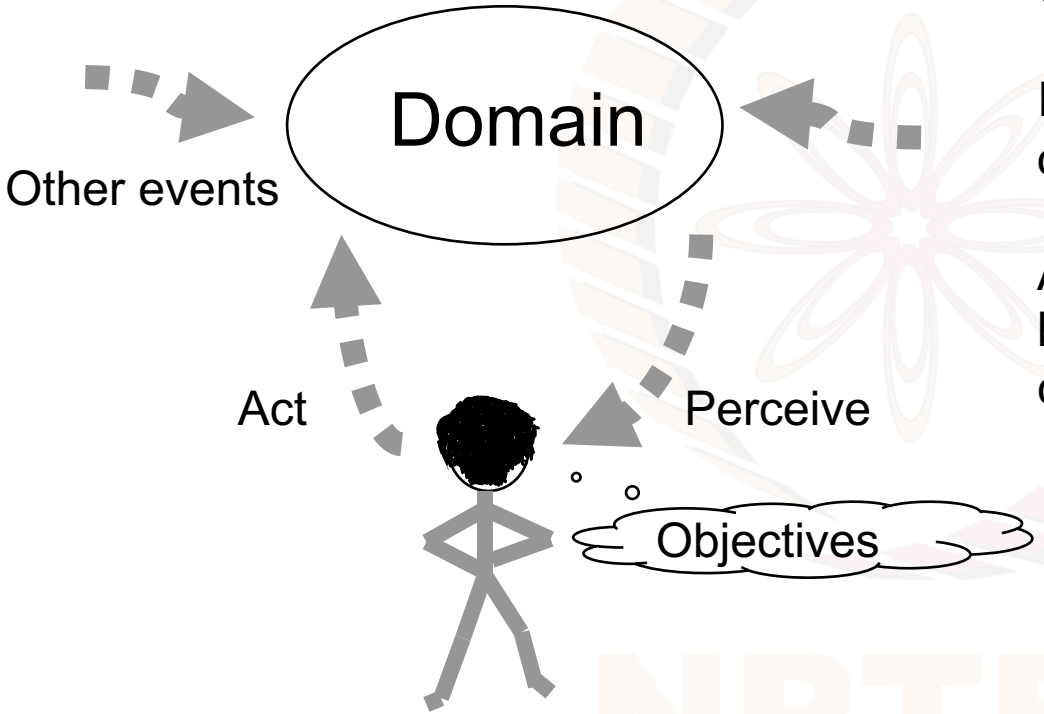
- So far our view of *problem solving* using search has been *state centered*
  - the *state space* is the arena for search
  - the *solution* is expressed as a *sequence of states*
  - even when we talk of solution space, the solution, for example for the TSP problem, is expressed in terms of states.
- The *planning community* takes an action centric view of problem solving
  - *even* when we use the phrase *state space planning*
  - solutions are expressed as *sequences of actions*
  - actions are still *applicable* to states
  - or sometimes *relevant* to goal descriptions
  - *plan space planning* represents *only* actions structures

The *Reasoning* side of *Acting*

- An autonomous agent in some domain
  - may have certain goals to achieve
  - may have access to a repository of actions or operators
  - may use search or other methods to find a plan
  - executes the plan and monitors it as well

- Goal driven behavior by an agent
  - *perceive* or sense its environment
  - *deliberate* – find a plan to achieve goals
  - *act* – execute the planned actions

Planning: A Quick Introduction



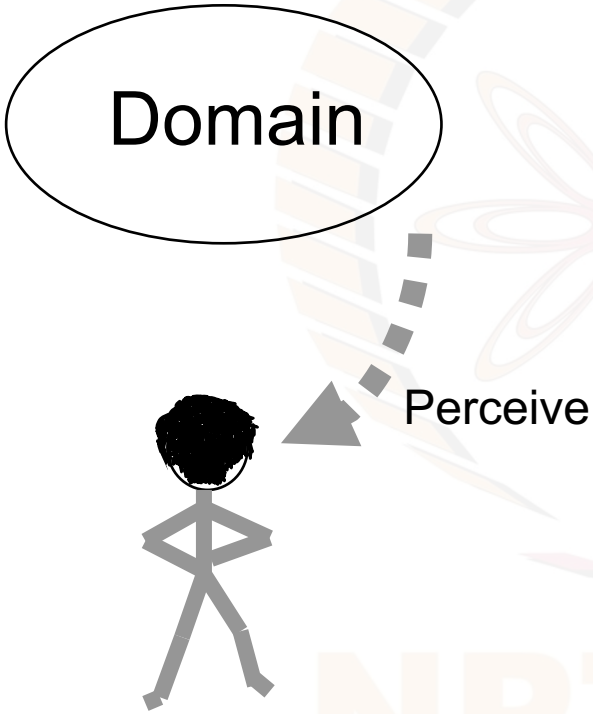
A planning agent can perceive the world, and produces actions designed to achieve its objectives.

In a static domain the agent is the only one who acts.

A dynamic domain can be modeled by including other agencies that can change the world.

Domain Independent Planning

Planning Domains: Perception

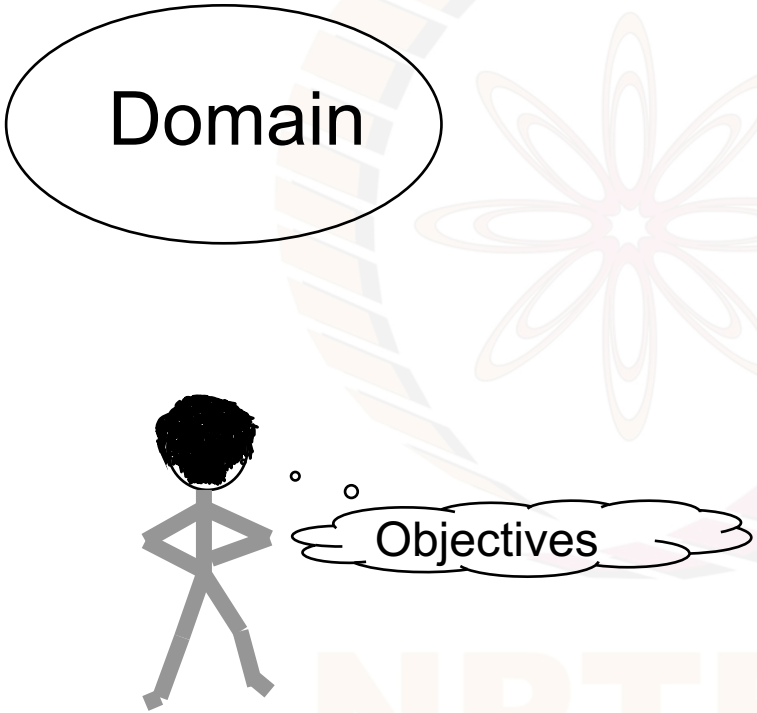


Domains can be modeled with various degrees of expressivity

In the *simplest domains* the agent can perceive the world perfectly - *complete information* domains

In *realistic* situations the agent may have *only partial information*

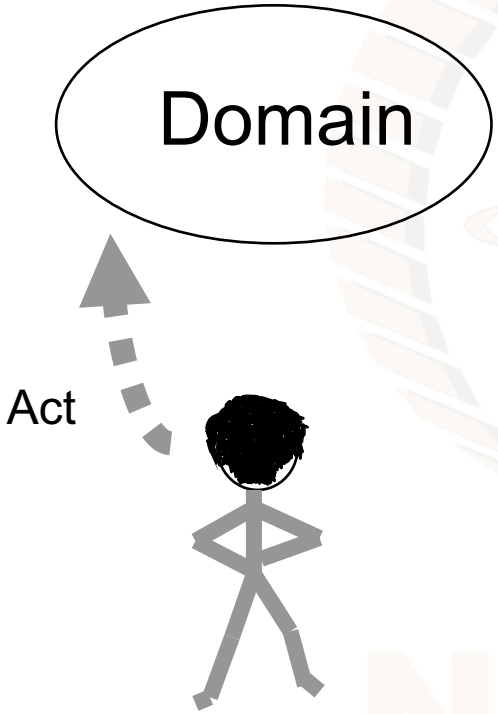
Planning Domains: Objectives



The goals or objectives of an agent can be of different kinds

- satisfaction goals on end state
  - *have* to be achieved
- soft constraints on end state
  - may not *all* be achieved
- hard *trajectory* constraints
  - not just the final state
  - on the *path* too
- soft trajectory constraints

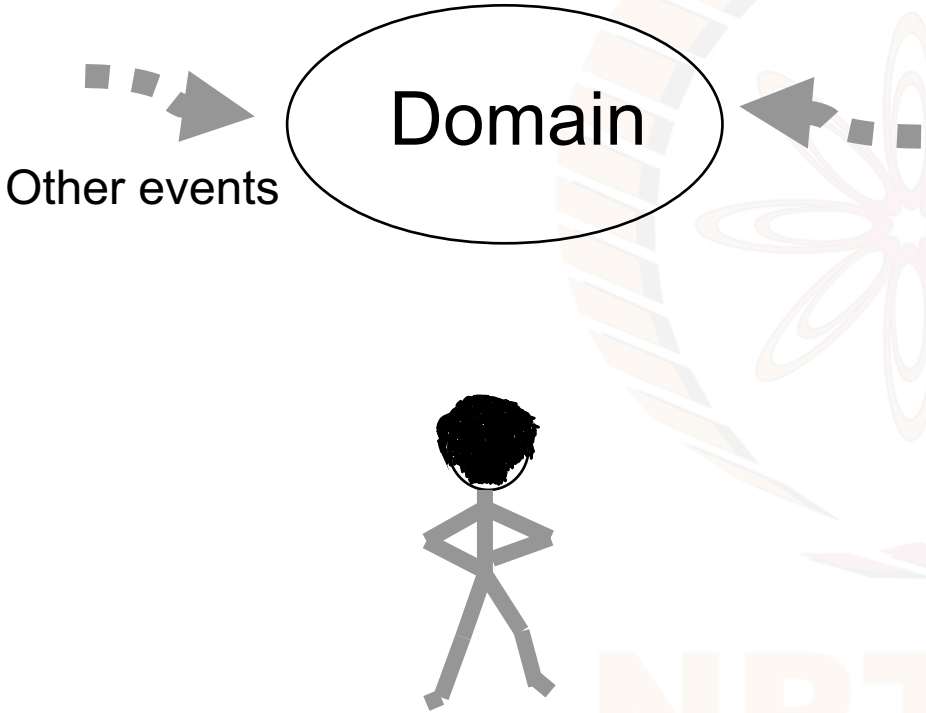
Planning Domains: Actions



Actions may be

- *deterministic*
  - *always* achieve the intended results
- *stochastic*
  - may or *may not* achieve the intended results
- *instantaneous*
  - no notion of time
- *durative*
  - have starting and ending time
- actions may have associated cost

Planning Domains: non solo



A planning agent may be the only one making changes in the world

There may extraneous events
- for example rain
- shops opening at certain times

Multi-agent systems
- there may be collaborating agents
- or adversarial agents
- or agents that may be competing for common resources

STRIPS domains

The simplest domains are called STRIPS domains

- STandford Research Institute Planning System

- Finite, static, completely observable environment.
  - the sets of states and actions is finite,
  - changes occur only in response to actions of agents
  - the agent has complete information
  - there are no other agents
- The goals are hard constraints on the final state
  - they have to be achieved in a valid plan
- Actions are instantaneous
  - there is no explicit notion of time
- Actions are deterministic
  - no accidents, execution errors, or stochastic effects

State Transition Systems

The simplest domains can be modelled as a *state-transition system* which is defined as a triple = (S, A, γ), where

- S is a finite set of *states* in which the system may be.
- A is a finite set of *actions* that the actor may perform.
- γ : $S \times A \rightarrow S$ is a partial function called the *state transition function*.
  - If action a is *applicable* in state s,
  - then γ(s, a) is the resulting state

Planning Domain Description Languages (PDDL)

In modern times a series of languages with increasing expressivity has been defined to bring uniformity to planning research

The idea is that the researchers working on planning algorithms can use these *standardized* representations

Components of a PDDL planning task:

- Objects: Things in the world that interest us.
- Predicates: Properties of objects that we are interested in.
- 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.

The family of PDDL languages

PDDL 1.0 – essentially the STRIPS domains

PDDL 1.2 – conditional effects

PDDL 2.1 – numerical fluents (for example to model fuel quantity)
– plan metrics
– durative actions

PDDL 2.2 – derived predicates (consequences of effects)
– timed initial literals (to model exogenous events)

PDDL 3.0 – state trajectory constraints
– preferences (soft constraints)

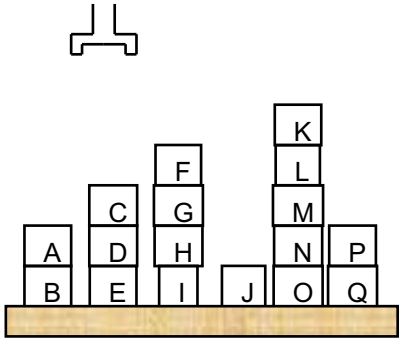
PDDL 3.1 – object fluents (functions could return objects)

We will confine ourselves to STRIPS domains

The Blocks Worlds domain

The world, in the blocks world domain, is described by a set of statements that conform to the following *predicate schema* –

on(X, Y) :	block X is on block Y
ontable(X):	block X is on the table
clear(X) :	no block is on block X
holding(X):	the robot arm is holding X
armempty:	the robot arm is not holding anything



- There are no metrics involved. Essentially a qualitative description
- A block can have only one block on it.
- We assume an arbitrarily large table.
- The one armed robot can hold one block.

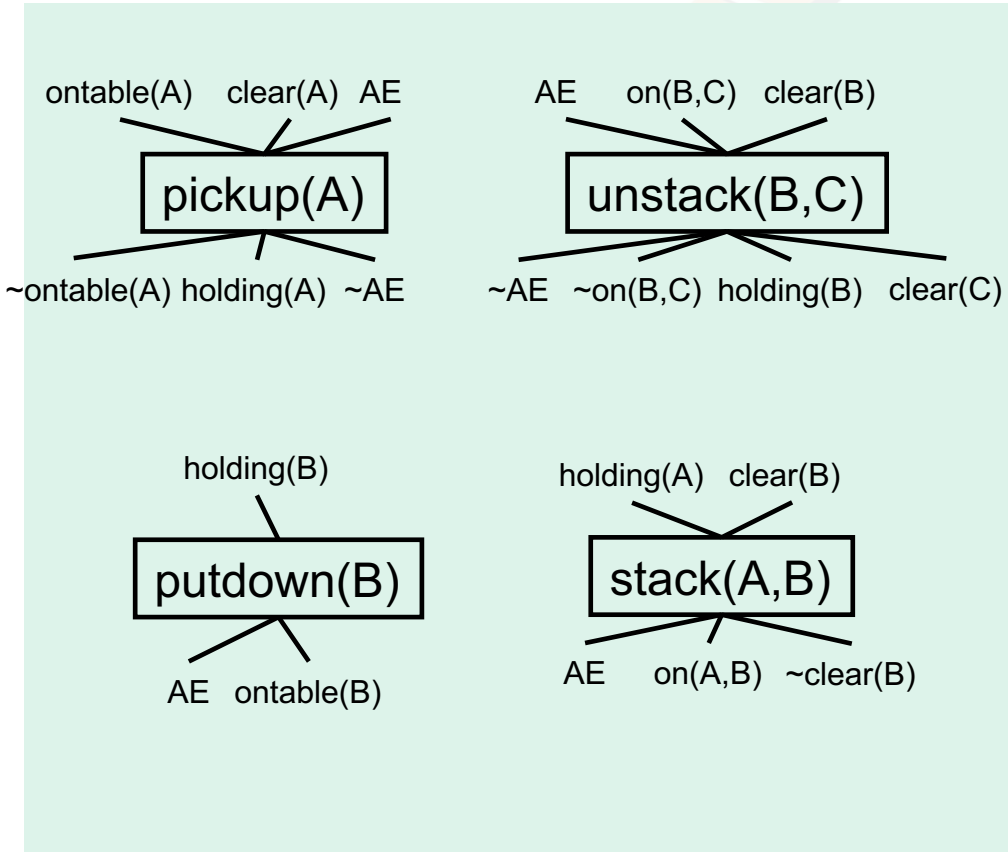
Operators and Actions

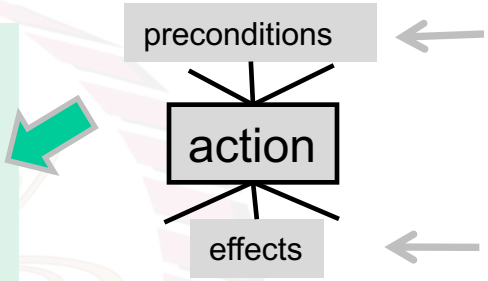
A planning *operator* O is defined by
- a name (arguments - object types)
- a set of preconditions: pre(O)
- a set of positive effects: $effects^{+}(O)$ → called the ADD list in STRIPS
- a set of negative effects: $effects^{-}(O)$ → called the DELETE list in STRIPS

An *action* is an instance of an operator
with individual objects as arguments

Also called ground operators

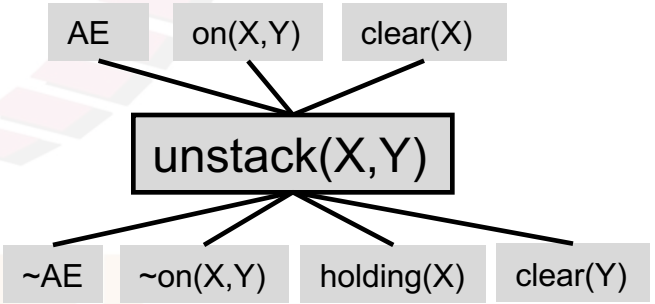
STRIPS Planning Domain: Blocks World



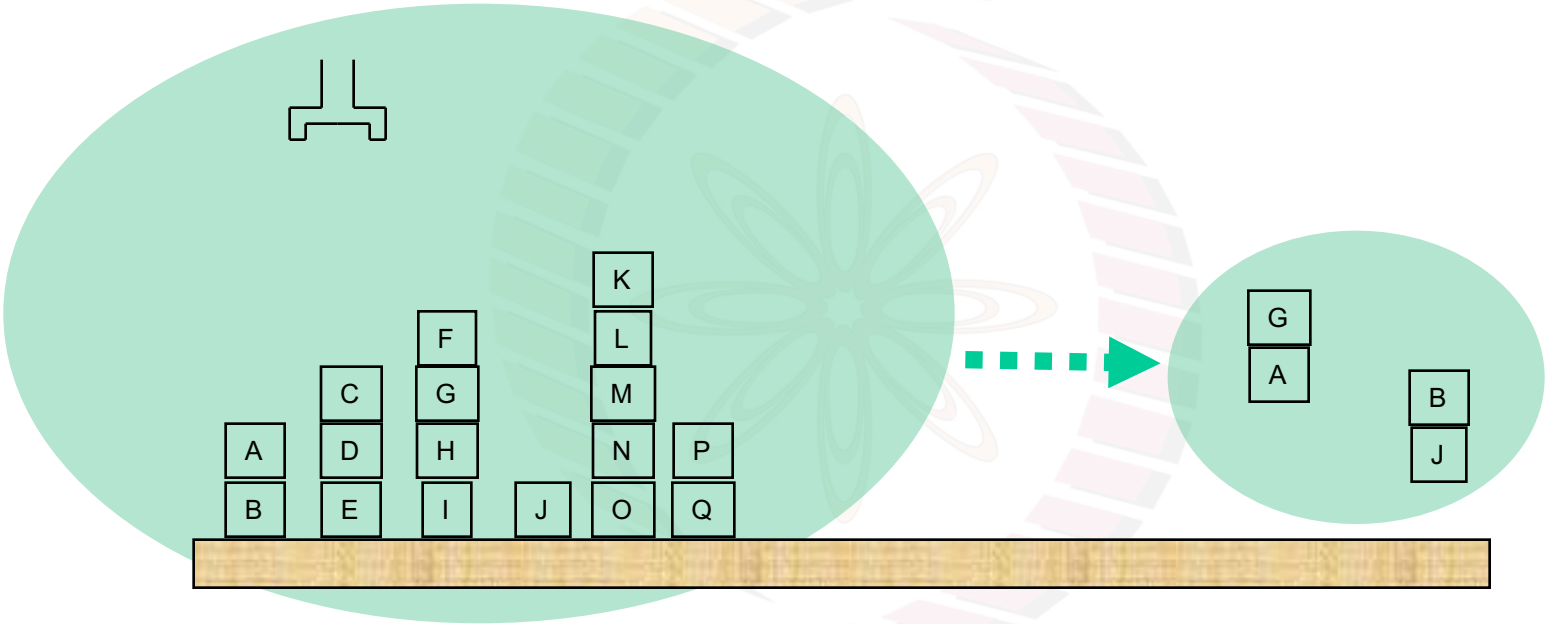


Predicates

ontable(X)
on(X,Y)
clear(X)
holding(X)
AE



State Space Planning

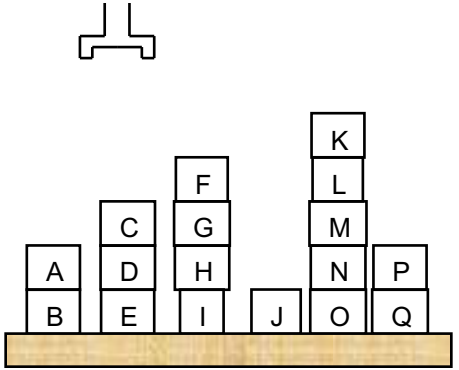


The Given State

The Goal = on(G,A) Λ on(B,J)

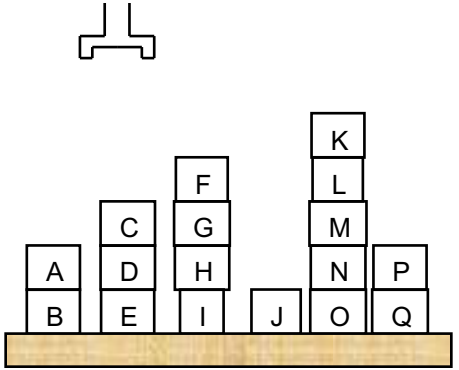
Note: The Goal is a partial state description.

The Given State is Completely Known



AE
Λ ontable(B) Λ on(A,B) Λ clear(A)
Λ ontable(E) Λ on(D,E) Λ on(C,D) Λ clear(C)
Λ ontable(I) Λ on(H,I) Λ on(G,H) Λ on(F,G) Λ clear(F) Λ ontable(J) Λ clear(J)
Λ ontable(O) Λ on(N,O) Λ on(M,N) Λ on(L,M) Λ on(K,L) Λ clear(K)
Λ ontable(Q) Λ on(P,Q) Λ clear(P)

The Given State is Completely Known



AE
Λ ontable(B) Λ on(A,B) Λ clear(A)
Λ ontable(E) Λ on(D,E) Λ on(C,D) Λ clear(C)
Λ ontable(I) Λ on(H,I) Λ on(G,H) Λ on(F,G) Λ clear(F) Λ ontable(J) Λ clear(J)
Λ ontable(O) Λ on(N,O) Λ on(M,N) Λ on(L,M) Λ on(K,L) Λ clear(K)
Λ ontable(Q) Λ on(P,Q) Λ clear(P)

Forward State Space Planning (FSSP)

Applicable actions: Given a state S an action *a* is *applicable* in the state if its preconditions are satisfied in the state. That is,

$$pre(a) \subseteq S$$

Progression: If an applicable action *a* is applied in a state *S* then the state *transitions* or *progresses* to a new state S', defined as,

$$S' = \gamma(S, a)$$
$$= \{S \cup effects^{+}(a)\} \setminus effects^{-}(a)$$

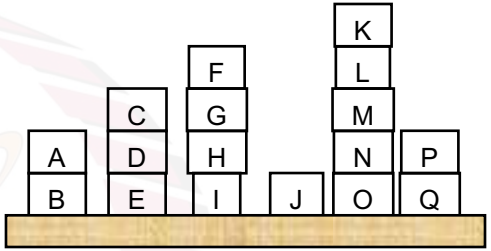
Plan: A plan π is a sequence of actions $<a_1, a_2, ..., a_n>$. A plan π is *applicable* in a state S_0 if there are states $S_1,...,S_n$ such that $\gamma(S_{i-1},a_i)=S_i$ for i =1,...,n.

The final state is $S_n= \gamma(S_0,\pi)$.

Valid plan: Let G be a goal description.
Then a plan π is a valid plan in a state state S_0 if,

$$G \subseteq \gamma(S_0,\pi)$$

Forward State Space Search



The Given State

Unstack(A,B)

Unstack(C,D)

Unstack(F,G)

Unstack(P,Q)

Putdown(F)

Pickup(J)

Unstack(K,L)

Stack(F,A)

Stack(F,C)

Stack(F,G)

Stack(F,J)

Stack(F,K)

Stack(F,P)

Huge tree!

Forward vs. Backward State Space Planning

FSSP has a *high branching factor*.

This is because the given state is *completely described* and *many actions* are applicable.

Without the guidance of a good heuristic function search may be computationally expensive.

On the other hand the *goal description is often quite small*

- for example G = (on(G,A) Λ on(B,J)) in the example

Will working backwards from the goal be more efficient?

- only the *relevant* actions will be considered

Let us explore this possibility...

Backward State Space Planning (BSSP)

Relevant actions: Given a goal G an action *a* is *relevant to the goal* if it produces some positive effect in the goal, and deletes none. That is,

$$\{\text{effect}^{+}(a) \cap G\} \neq \phi \ \wedge \{\text{effects}^{-}(a) \cap G\} = \phi$$

Regression: If a relevant action *a* is applied in a goal G then the goal *regresses* to a new goal G', defined as,

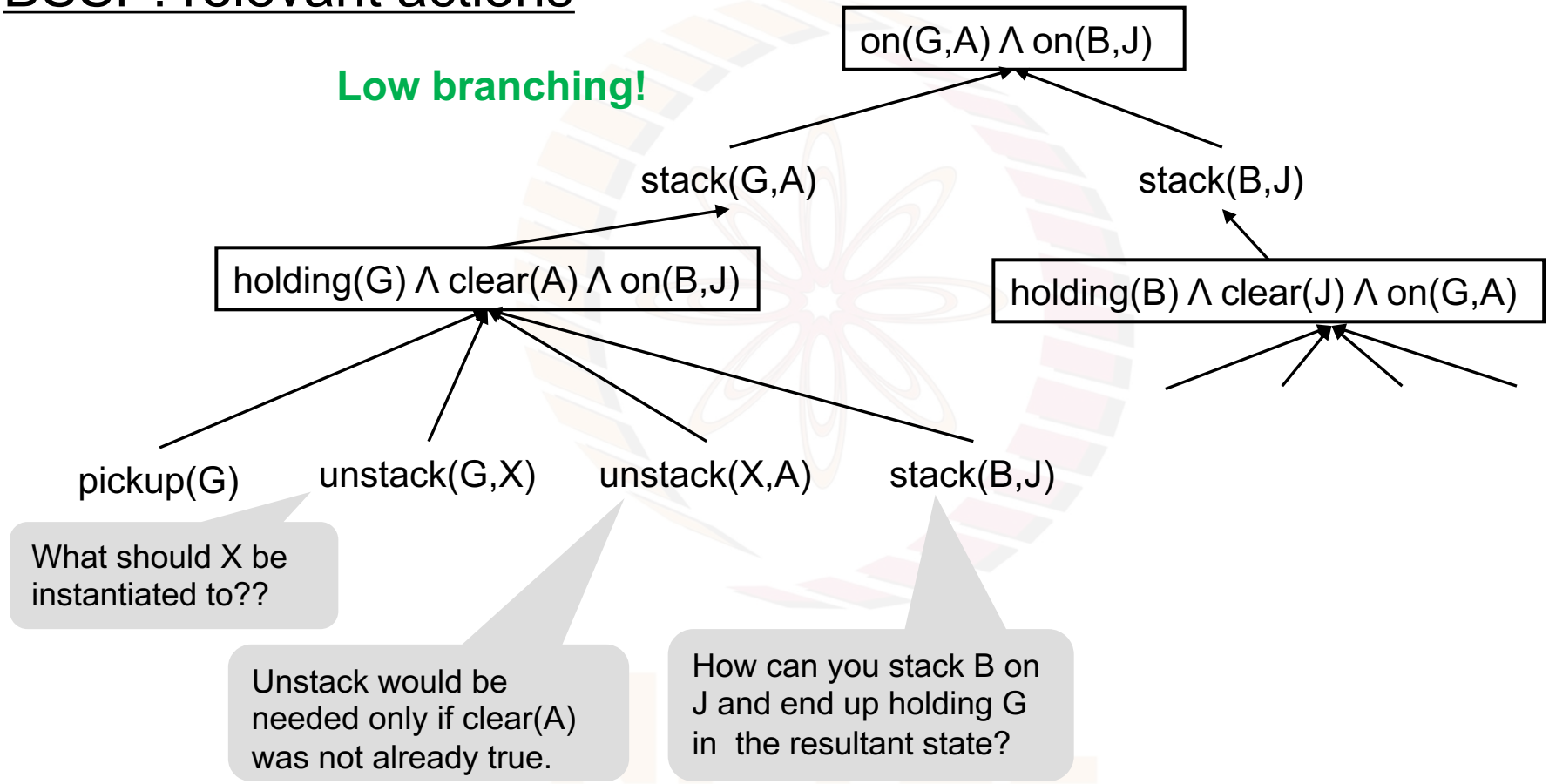
$$G' = \gamma^{-1}(G, a)$$
$$= \{G \setminus \text{effects}^{+}(a)\} \cup \text{pre}(a)$$

Plan: A plan π is a sequence of actions $<a_1, a_2, ..., a_n>$. A plan π is *relevant* in to a goal G_n if there are goals $G_1,...,G_{n-1}$ such that $G_{i-1} = \gamma^{-1}(G_i, a_i)$ for $i = 1,...,n$. The final goal is $G_1 = \gamma^{-1}(G_n, \pi)$

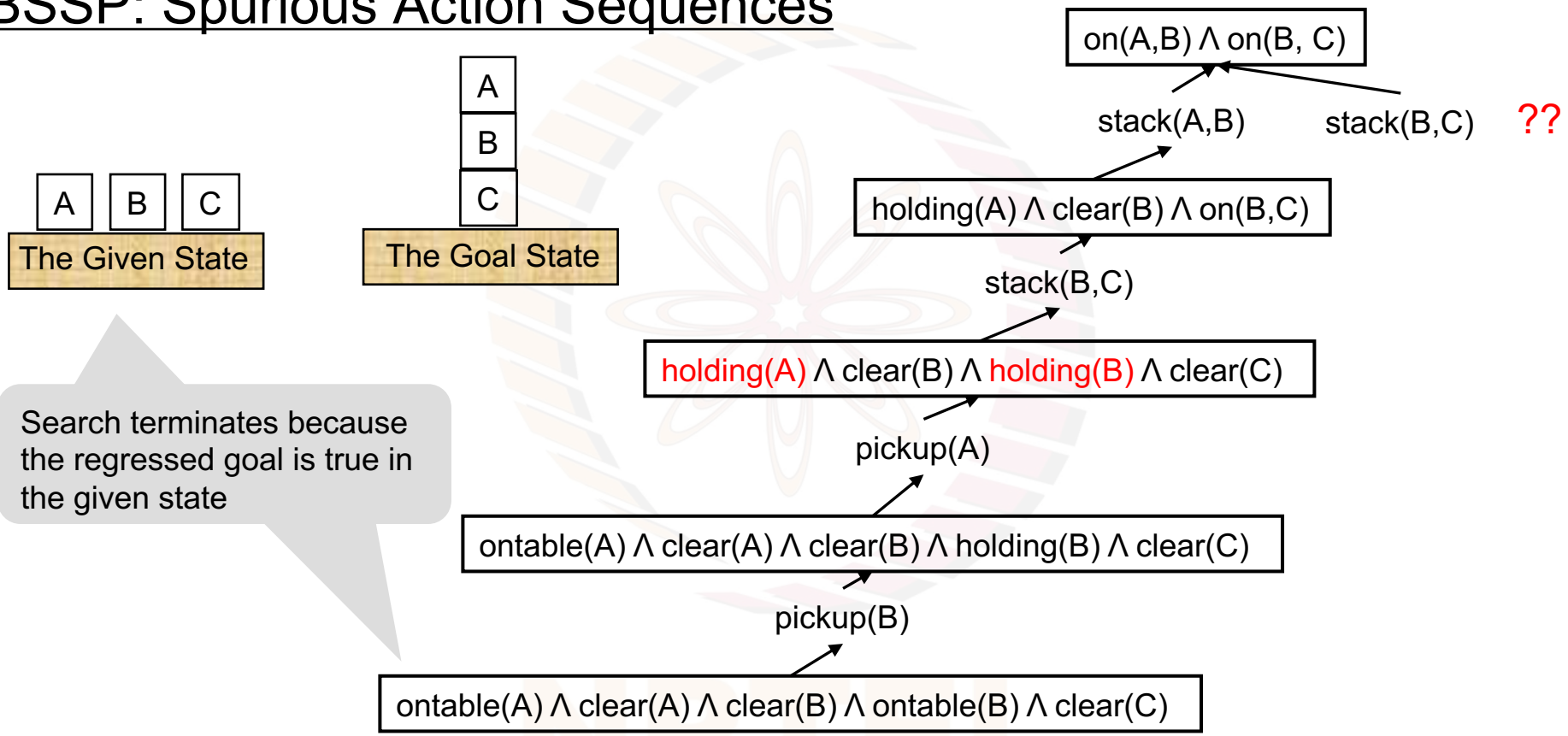
Valid plan: Let S_0 be the start state. Regression ends when $G_1 \subseteq S_0$
The plan π is a valid plan if $G_n \subseteq \gamma(S_0, \pi)$

note: validity still checked by progression

BSSP: relevant actions



BSSP: Spurious Action Sequences



<pickup(B), pickup(A), stack(B,C), stack(A,B)> is not a valid plan.

Forward and Backward: Exploring the space

	FSSP	BSSP
Start at	Start State	Goal Description
Moves	*a* is *applicable* $pre(a) \subseteq S$	*a* is relevant $\{effect^{+}(a) \cap G\} \neq \phi \wedge \{effects^{-}(a) \cap G\} = \phi$
Transition	Progression $S' = \{S \cup effects^{+}(a)\} \setminus effects^{-}(a)$ Sound $\pi \leftarrow \pi \circ a$	Regression $G' = \{G \setminus effects^{+}(a)\} \cup pre(a)$ not Sound $\pi \leftarrow a \circ \pi$
GoalTest	$G \subseteq \gamma(S_0, \pi)$	$\gamma^{-1}(G_n, \pi) \subseteq S_0$ followed by validity check

Goals: some terminology

The start *state* and the goal *description* are described as *sets of sentences* expressed as *predicates* in *first order logic**

The predicates, like *on(A,B)* and *holding(D)* in the blocks world, are defined in the specific PDDL language used to *define the domain*.

We often refer to these predicates as *goals*, in the spirit of *goal-directed planning* in which *each of them is a goal that needs to be true in the goal state.*

The goal of planning G = $\{g_1, g_2, g_3\}$

- for example {on(A,B), on (B,C)}
- a *partial description of a state*
- represents a *set of states* containing the goals.

A *state* is a set in which *all the goals are true*.

For an in depth study of logic see our course
- Artificial Intelligence: Knowledge Representation & Reasoning

Planning operators: the arrow of time

A planning operator is defined with the *arrow of time* built in implicitly.

- It is essentially *designed to progress in the state space.*

When an *applicable* action *a* is applied in a state S

- effect^{+}(a) are added to S, because they *become true*
- effect^{-}(a) are deleted from S, because they *become false*
- the resulting set of goals S' define a *valid state*
- progression is *closed* over the state space

A *relevant* action *a* addresses one or more goals in G

- regression over *a* adds the pre(a) to G
- and *carries backwards* the *other goals*
- but the resulting set G' *may not be feasible*
- *e.g.* {holding(A), clear(B), holding(B), clear(C)}
- regression is *not closed* over the state space
- BSSP *requires a check* and possibly extra work

Goal Stack Planning (GSP)

FSSP - searches for *applicable actions* from the start state S
- constructs the plan from the start state
- the *first action found* is the *first action in the plan*
- suffers from *high branching* since S is a full description

BSSP - searches for *relevant actions* from the goal description G
- construct the plan from the the goal description
- the *first action found* is the *last one in the plan*
- suffers from *spurious sub-goals*

Goal Stack Planning is an algorithm designed to
- *search* in a *goal directed backward* manner
- to take advantage of low branching
- but *construct plans in a forward* manner
- adding actions that are *applicable*

GSP: Linear Planning

The basic idea of goal stack planning is to

- break up a compound goal into individual goals
- and solve them serially one by one
- producing a plan that is a sequence of actions
- a linear plan

It is best suited to domains where goals

- can be solved serially one at a time
- for example, cook three dishes
- each goal can be solved independently

GSP employs a stack in which

- goals are pushed in (in a backward manner)
- along with actions that could achieve them

Function PushSet(G)

A basic operation in GSP is to

- push a compound goal G = {g_1, g_2, ..., g_n}
- along with the individual goals g_1, g_2, ..., g_n
- to be solved in *last in first out* order
- possibility of using a heuristic function
- or some form of reasoning

The reason one has to insert the compound goal too is to check whether after having solved the individual goals independently the compound goal has indeed been solved

- we will illustrate this need with an example

When a goal is popped from the stack...

The algorithm goal stack planning maintains the current state S at all times.

When a goal *g* is popped from the stack, there are two possibilities

1. $g \in S$

   When the goal is true in the current state nothing needs to be done.

2. $g \notin S$

   When the goal is not true

   - GSP pushes a relevant action *a* onto the stack
   - Followed by PushSet(pre(a))
   - i.e. the *precondtions of a* as a compound goal
   - and the individual goals in pre(a) in some order

If an action *a* is popped then

it *must be applicable* and can be added to the plan

- $\pi \leftarrow \pi \circ a$
- $S' \leftarrow \text{Progress}(S,a) = \{S \cup \text{effects}^+(a)\} \setminus \text{effects}^-(a)$

Algorithm GSP

```
GSP(givenState, givenGoal, actions)
1   S ← givenState ; plan ← ( ) ; stack ← emptyStack
2   PushSet(givenGoal, stack)
3   while not Empty(stack)
4         do x ← Pop(stack)
5         if x is an action a
6            then  plan ← (plan ∘ a)
7                  S ← Progress(S, a)
8         else if x a compound goal G and G is not true
9            then  PushSet(G, stack)
10        else if x is a goal g and g ∉ S
11           then  CHOOSE a relevant action a that achieves g
12                 if none then return FAILURE
13                 Push(a, stack)
14                 PushSet(Pre(a), stack)
15  return plan
```

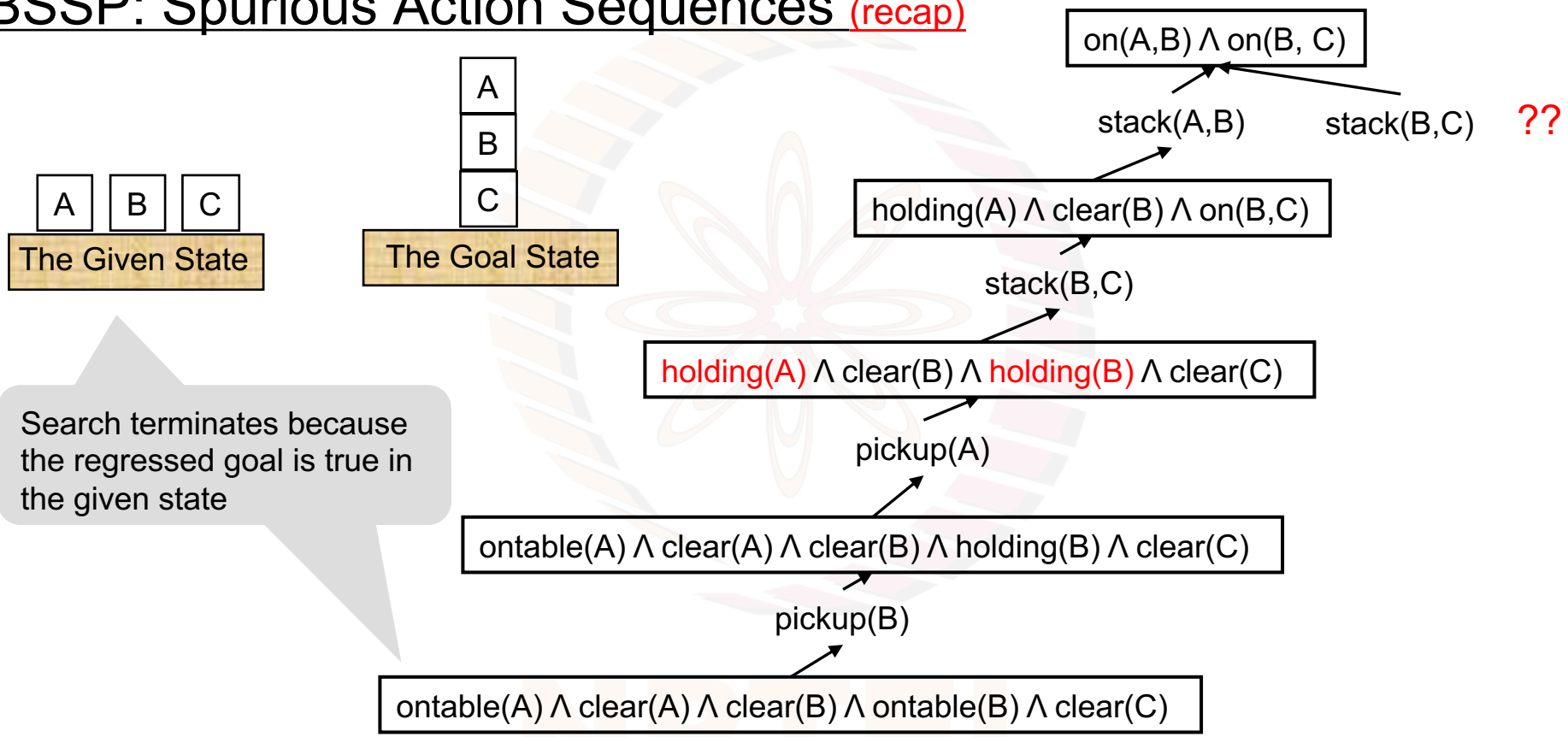
A planning problem

Backward search

Forward plan construction

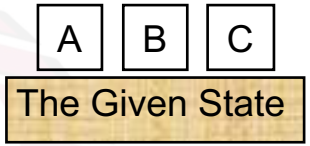
Non-deterministic choice - replace with backtracking

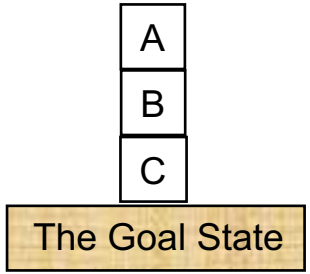
BSSP: Spurious Action Sequences (recap)



<pickup(B), pickup(A), stack(B,C), stack(A,B)> is not a valid plan.

GSP example: correct goal order





stack

{on(A,B), on(B,C)} — PushSet

on(A,B) — Pop – not true

~~on (B,C)~~ — Push relevant action

stack(B,C)

{holding(B), clear(C)} — PushSet pre(stack(B,C))

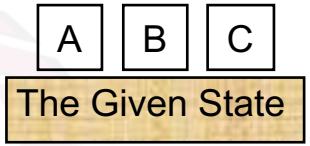
~~holding(B)~~ — Pop – true

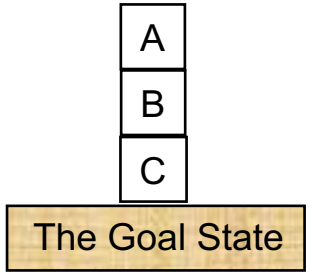
~~clear(C)~~

pickup(B) — Pop – not true

Push relevant action and preconditions

GSP example: correct goal order





stack

{on(A,B), on(B,C}

on(A,B)

stack(B,C)

{holding(B), clear(C)}

pickup(B)

~~{onT(B), clear(B), AE}~~

~~onT(B)~~

~~clear(B)~~

~~AE~~

Push relevant action and preconditions

Pop - true

Pop - true

Pop - true

Pop - true

Now pop action pickup(B)
...the first action in the plan

GSP example: correct goal order

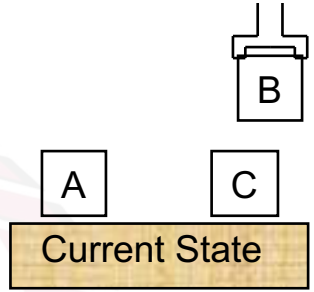
stack

{on(A,B), on(B,C}

on(A,B)

~~stack(B,C)~~

~~{holding(B), clear(C)}~~



Plan = <Pickup(B)>

Pop compound action - true

Next pop action stack(B,C))

Plan = <Pickup(B)>, Stack(B,C)>

GSP example: correct goal order

stack

{on(A,B), on(B,C)}

on(A,B)

Pop – not true

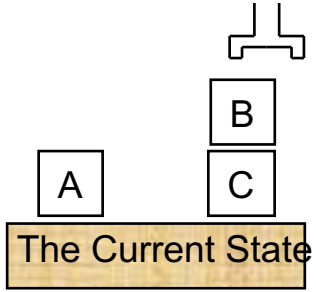
Stack(A,B)

{holding(A), clear(B)}

Push relevant action and preconditions

holding(A)

clear(B)



Plan = <Pickup(B)>, Stack(B,C)>

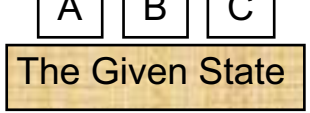
B is clear,
GSP will next find the action PickUp(A) to make holding(A) true
This will be the third action, and Stack(A,B) the last action
Final plan = <Pickup(B)>, Stack(B,C), Pickup(A), Stack(A,B)>

GSP example: wrong goal order

~~{on(A,B), on(B,C)}~~

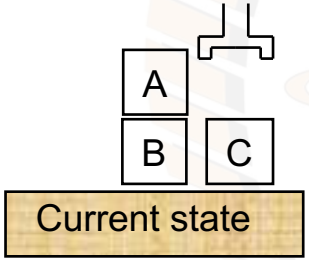
~~on(B,C)~~

~~on (A,B)~~



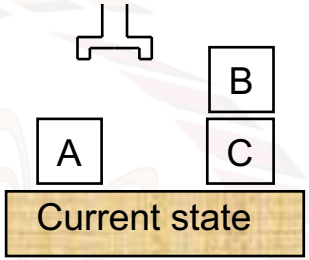
First achieve on(A,B)

Pickup(A)
Stack(A,B)



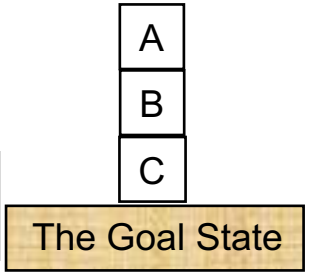
..then Achieve on(B,C)

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



Achieve on(A,B) again!

Pickup(A)
Stack(A,B)



A longer plan

Goal ordering matters

We saw in the tiny example that goal order matters

In this domain the wrong goal order resulted in a longer plan

In other domains a wrong goal order may lead to a dead-end

- for example during cooking
- requiring backtracking to a different goal order

So, does it mean that choosing a correct goal order will always result in an optimal plan?

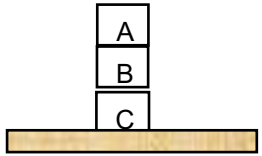
No!

Certain problems have *non-serializable subgoals*!

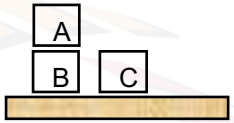
- there is no optimal order for solving them
- solving one may *undo* a previously achieved goal
- for example in the 8-puzzle and the Rubik's cube
- Gerald Sussman gave us a tiny example!

Sussman's Anomaly

{on(A,B), on(B,C)}

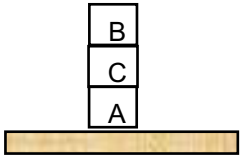


The Goal State



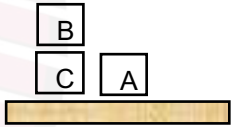
Then achieve on(A,B)

3.unstack(B,C)
4.putdown(B)
5.unstack(C,A)
6.putdown(C)
7.pickup(A)
8.stack(A,B)



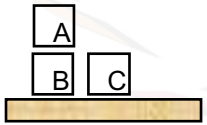
First achieve on(B,C)

1.pickup(B)
2.stack(B,C)

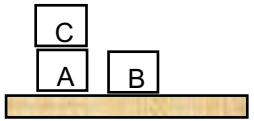


Then achieve on(B,C)

5.unstack(A,B)
6.putdown(A)
7.pickup(B)
8.stack(B,C)



First achieve on(A,B)



The Given State

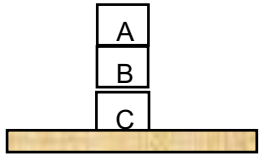
1.unstack(C,A)
2.putdown(C)
3.pickup(A)
4.stack(A,B)

Neither works!

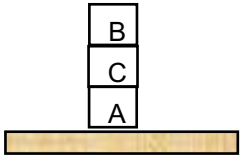
Sussman's Anomaly – optimal solution

Sussman's Anomaly – non-serializable subgoals

{on(A,B), on(B,C)}
on(B,C)
on (A,B)

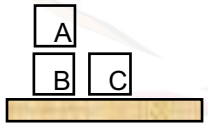


The Goal State

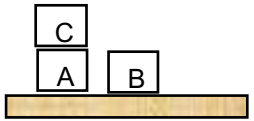


First achieve on(B,C)

3.pickup(B)
4.stack(B,C) → Shift back to on(A,B)



First achieve on(A,B)



The Given State

1.unstack(C,A)
2.putdown(C) → Shift focus to on(B,C)
5.pickup(A)
6.stack(A,B)

Next

Non-linear planning

Plan Space Planning

The planning approaches described so far reason with *states*.

The planner is basically looking at a *state* and a *goal description*.

If the *state satisfies the goal description* then it terminates.

Otherwise it searches *over the state space*
looking for actions to add to the plan.
- the plan is a by product of state space search

An alternative view is
to consider the *space of all possible plans*,
and search in *this* space for a plan.

We will call such approaches as *plan space planning*.

Partial plans

The search space in plans space planning (PSP) constitutes of *partial plans*

A partial plan π is a 4-tuple

$\pi = \langle A, O, L, B \rangle$ where

- A is the set of partially instantiated operators in the plan,
- O is the set of ordering links or relations of the form $(A_i \prec A_j)$,
- L is the set of causal links of the form (A_i, P, A_j),
- B is the set of binding constraints of the form -

$(?X = ?Y)^*$, $(?X \neq ?Y)$, $(?X = A)$ or $(?X \neq B)$ or $(?X \in D_X)$

where D_X is a subset of the domain of ?X.

* In the style often used in First Order Logic we use the prefix '?' to distinguish a variable '?X' from a constant such as 'A'. In some literature characters such as X, Y, and Z are reserved for variables while A,B, and C are reserved for constants. The convention we use makes programming simpler.

Partial plans

The constituents of a partial plan $\pi = \langle A, O, L, B \rangle$ are

- A is the set of partially instantiated operators in the plan,
  - The set *simply identifies* the actions that are in the plan. The actions may be *partially instantiated*, for example, Stack(A,?X) – stack block A onto *some block*
- O is the set of ordering links or relations of the form $(A_i \prec A_j)$,
  - The partial plan is thus a directed graph
  - It is also a partial order on actions in the plan
- L is the set of causal links of the form (A_i, P, A_j),
  - The causal link (A_i, P, A_j) represents fact that action A_i has a *positive effect* P which is a *precondition* for A_j
  - A_i is the *producer* of P, and A_j the *consumer* of P
  - When a causal link is added, so also is an ordering link
- B is the set of binding constraints of the form -

  $(?X = ?Y)$, $(?X \neq ?Y)$, $(?X = A)$ or $(?X \neq B)$ or $(?X \in D_X)$

  where D_X is a subset of the domain of ?X.

The Initial Plan

Given the planning problem

$<S=\{s_1, s_2, \ldots, s_n\}, G = \{g_1, g_2, \ldots, g_k\}, O\}$

Planning in *plan space planning*

always begins with an *initial plan*

$\pi_0 = <\{A_0, A_\infty\}, \{(A_0 \prec A_\infty)\}, \{\ \}, \{\ \}>$ where

A_0 is the *initial action* with *no preconditions* and

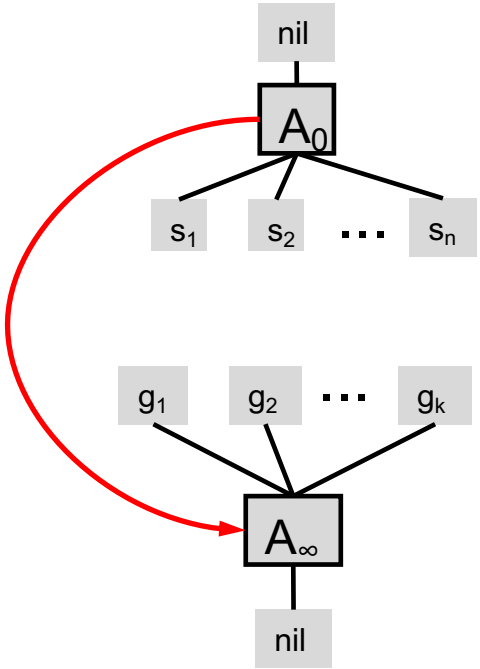
the *effects* $s_1, s_2, \ldots, s_n$

A_∞ is the *final action* with *preconditions* $g_1, g_2, \ldots, g_k$ and

no effects

$(A_0 \prec A_\infty)$ says that A_0 *precedes* A_∞

There are no *causal links* and *binding constraints*



Flaws

A partial plan may have two kinds of *flaws*

1. *Open goals* – any precondition of any action that is *not* supported by a causal link

2. *Threats* – A causal link (A_i, P, A_j) is said to have a *threat* if there exists another action A_t in the plan that potentially deletes *P*

Plan space planning involves systematically removing flaws

A *solution plan* is a partial plan *without any flaws*

Open goals

For any action to be *applicable* its *preconditions must be true.*

Any precondition P not supported by a causal link is an *open goal.*

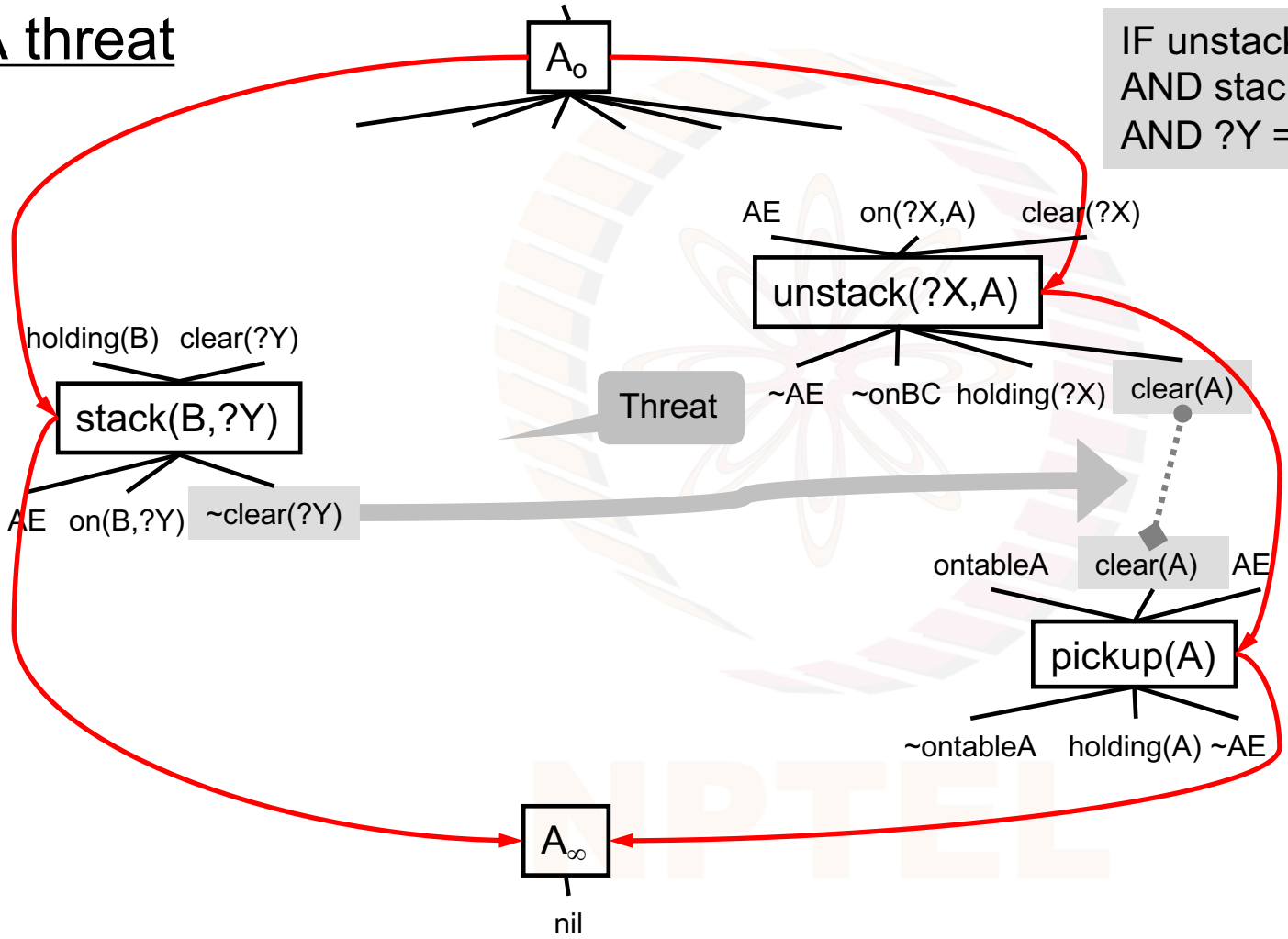
A causal link for P can be found in two ways.

1. If an *existing action* A_e produces P and it is *consistent* to add ($A_e \prec A_p$), then
   a) add a causal link (A_e, P, A_p) and
   b) add the ordering link ($A_e \prec A_p$) to the partial plan.

2. If *no such existing action* can be found then
   a) insert a new action A_{new} that produces P to the partial plan,
   b) add the corresponding causal link (A_{new}, P, A_p),
   c) and add the ordering link ($A_{new} \prec A_p$) to the partial plan.

A threat

IF unstack(?X,A) ≺ stack(B,?Y)
AND stack(B,?Y) ≺ pickup(A)
AND ?Y = A



Threats

An action A_{threat} that *can possibly* disrupt an existing causal link (A_i, P, A_j) is a *threat* to the link.

Disruption *will* happen *if all three of the following happen*:

(1) A_{threat} has an effect ~Q such the P can be unified* with Q.
(2) A_{threat} happens after A_i.
(3) A_{threat} happens before A_j.

If *all the three happen* then we say that the threat has *materialized.*

* Either P=Q or both P and Q are of the form $R(?X_1, \ldots, ?X_n)$ and the variables in P and Q can be unified. Please see Chapter 12 of *A First Course in Artificial Intelligence* for a detailed account of unification.

Resolving a threat

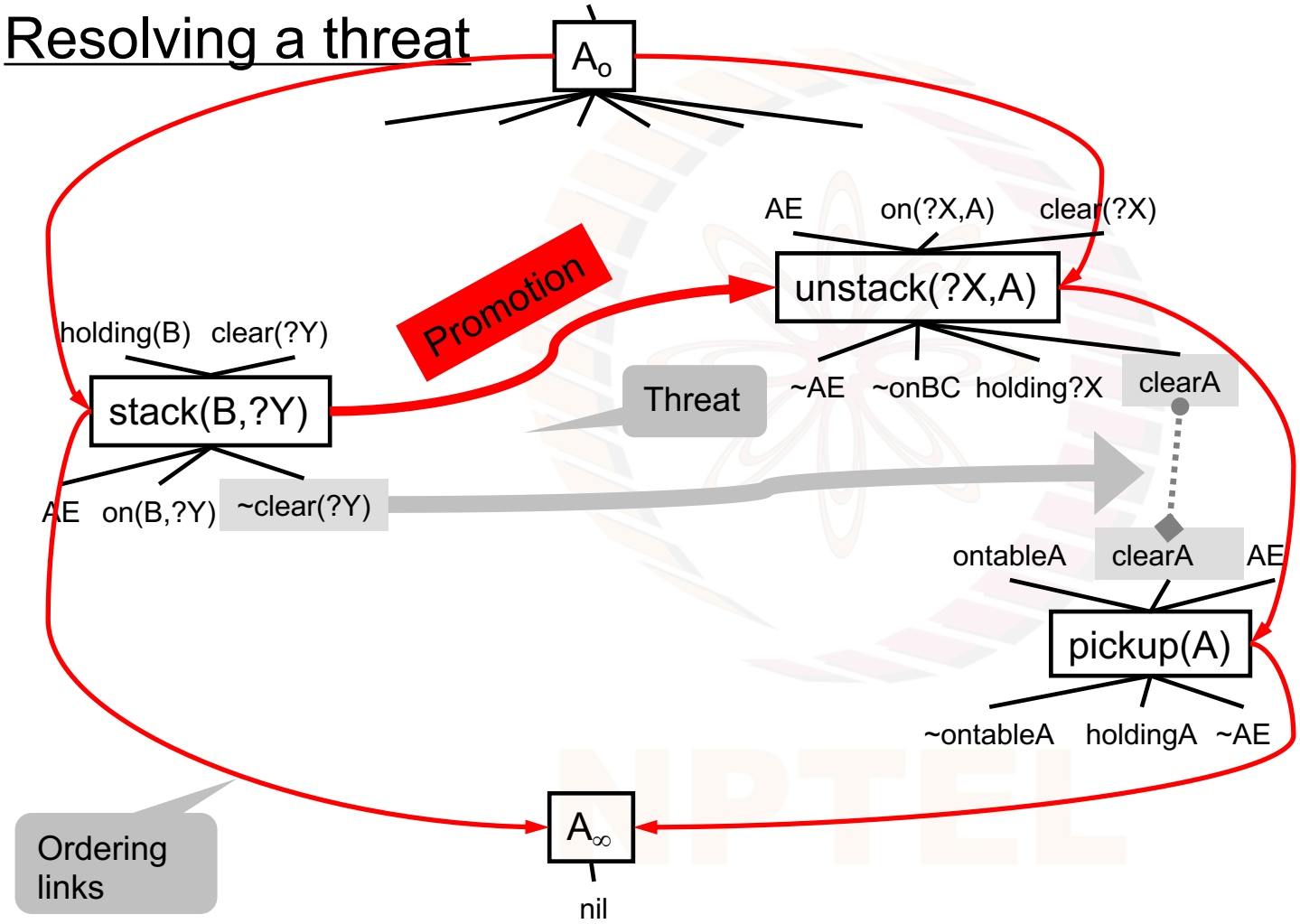
To eliminate the threat one needs to ensure that *at least one of the three conditions* for the threat is *not* met. This can be done by, respectively,

- *Separation*: Ensure that P and Q *cannot unify*.
  - This can be done by adding an *appropriate binding constraint* to the set B in the partial plan.

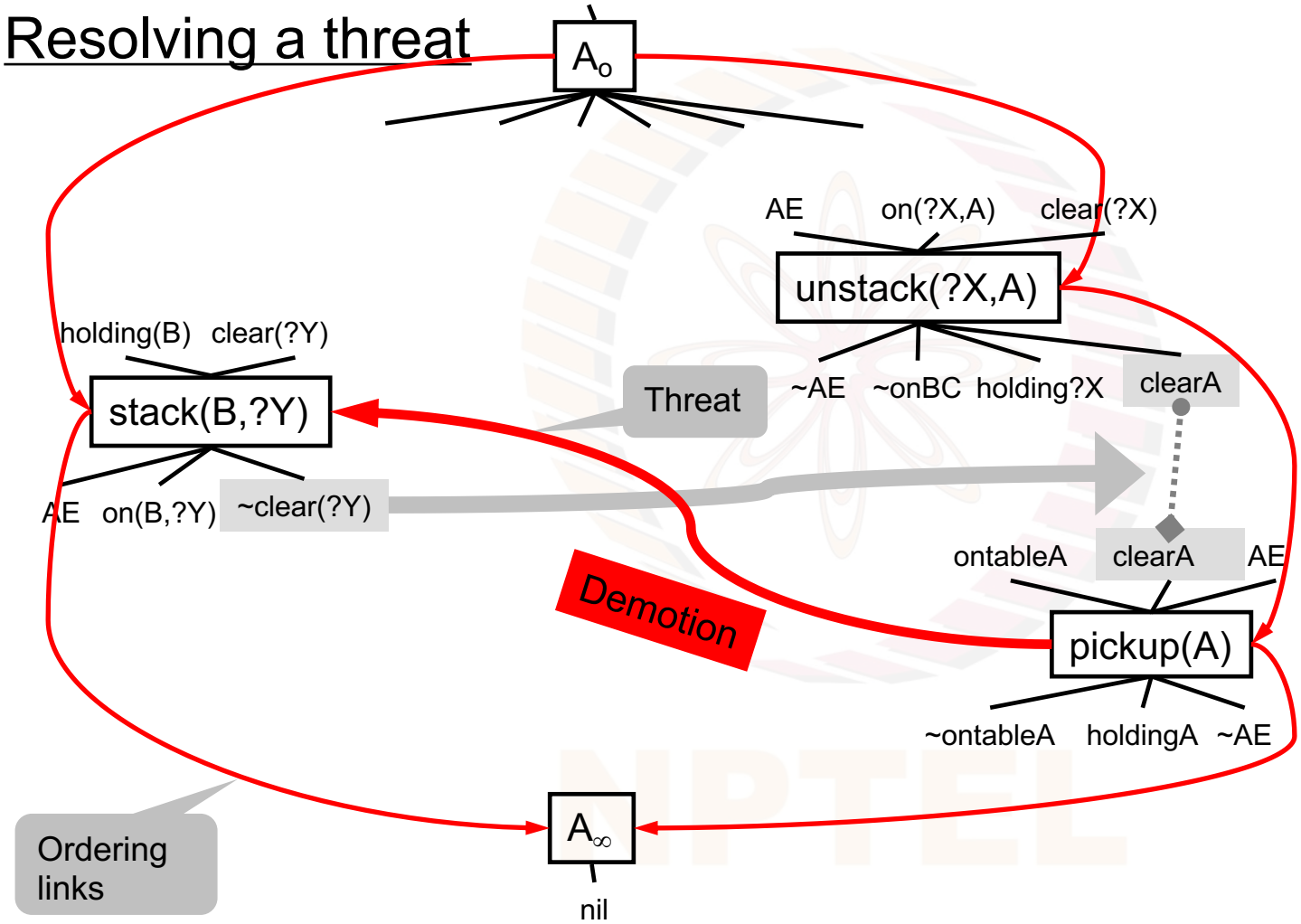
- *Promotion*: Advance the action A_{threat} to happen *before* it can disrupt the causal link.
  - Add an ordering link ($A_{threat} \prec A_i$) to the set O in the partial plan

- *Demotion*: Delay the action A_{threat} to happen *after* both the causal link actions.
  - Add an ordering link ($A_j \prec A_{threat}$) to the set O

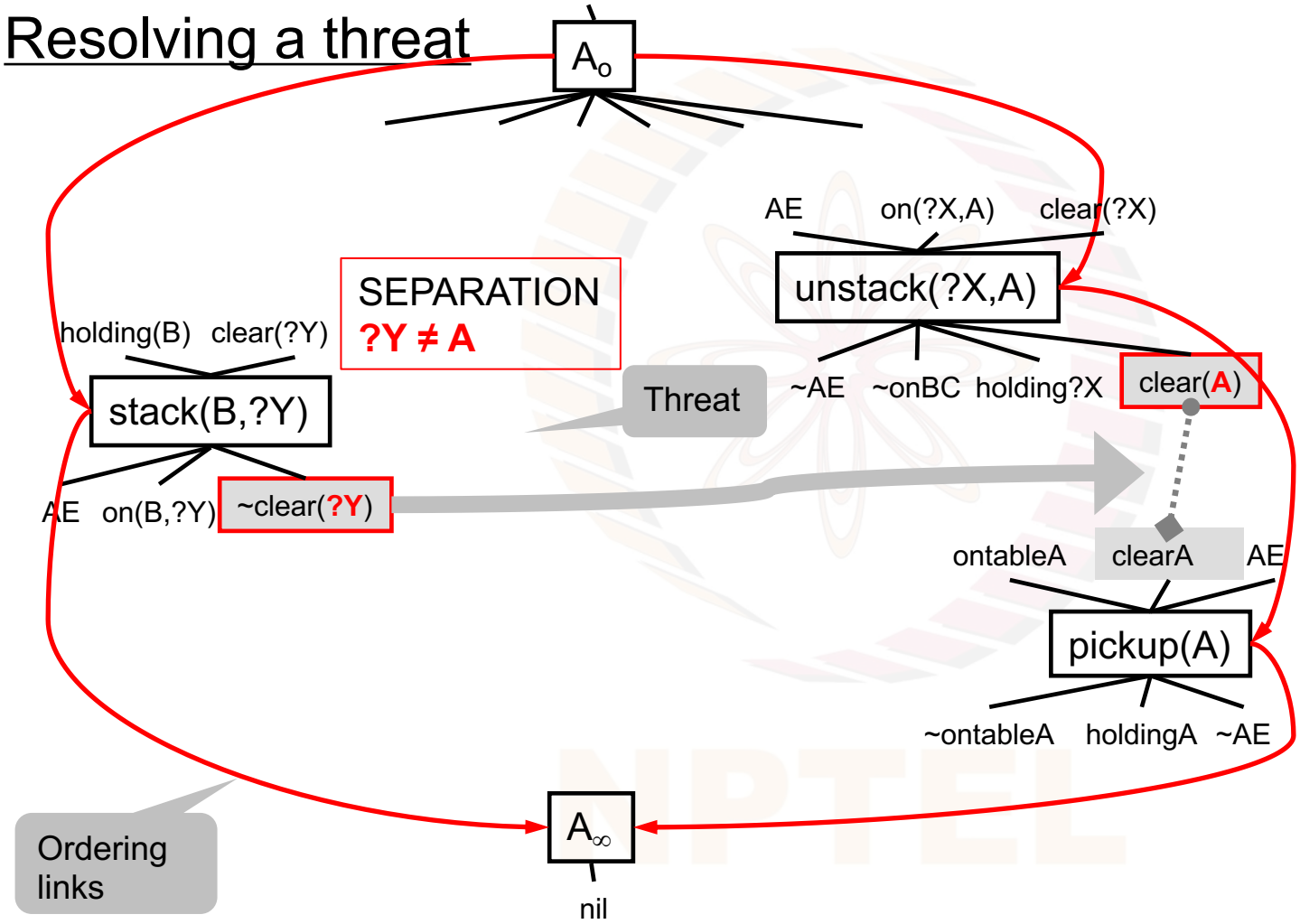
Resolving a threat



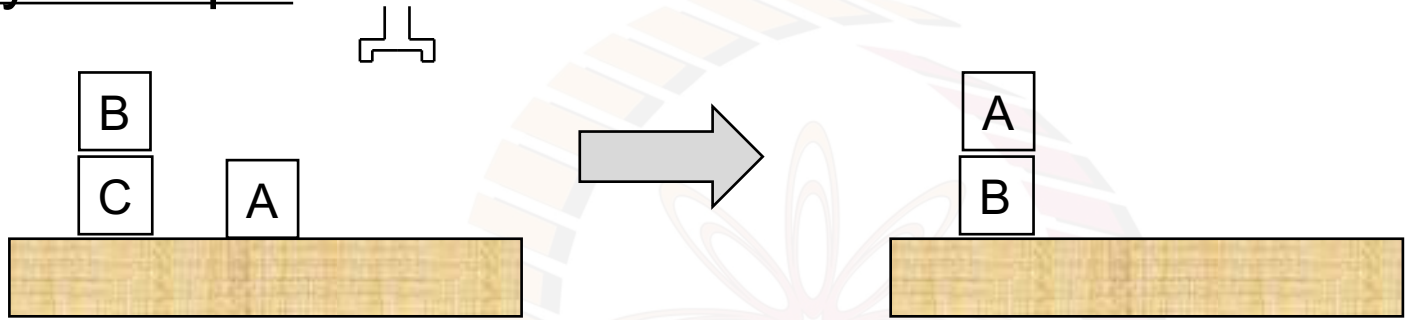
Resolving a threat



Resolving a threat



A tiny example



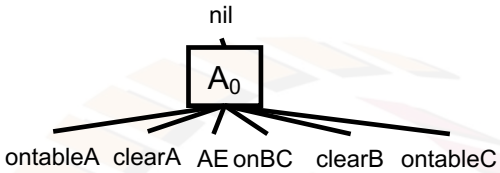
Start state

ontable(A)
ontable(C)
on(B,C)
clear(A)
clear(B)
AE

Goal description

ontable(B)
on(A,B)

The empty plan

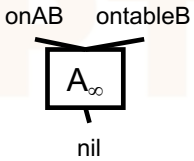


$\pi_0 = <\{A_0, A_\infty\}, \{(A_0 \prec A_\infty)\}, \{\}, \{\}>$

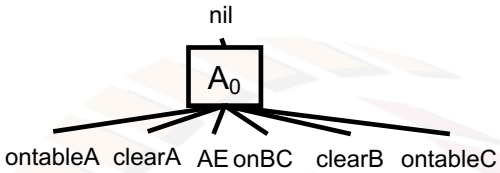
The ordering links are implicit

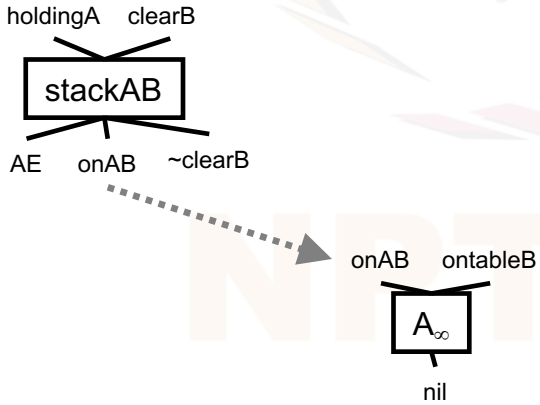
Unless shown explicitly actions higher up happen first

The empty plan stands for the set of all possible plans for this problem



The first action





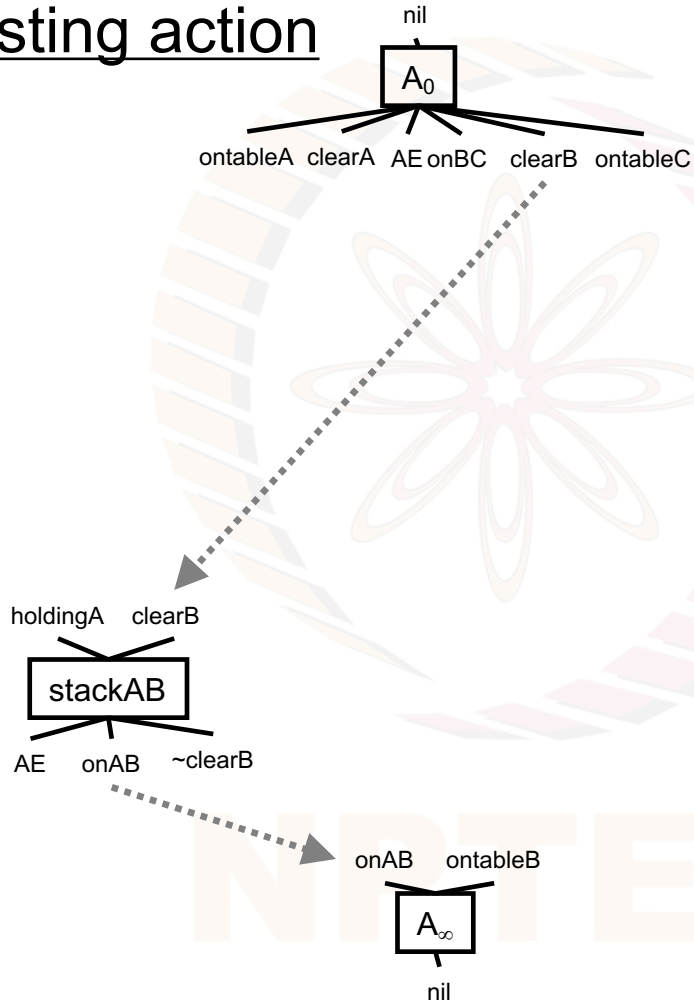
Action stack(A,B) provides a causal link
for the open goal on(A,B)

Not shown here -

$(\text{stack}(A,B) \prec A_\infty)$

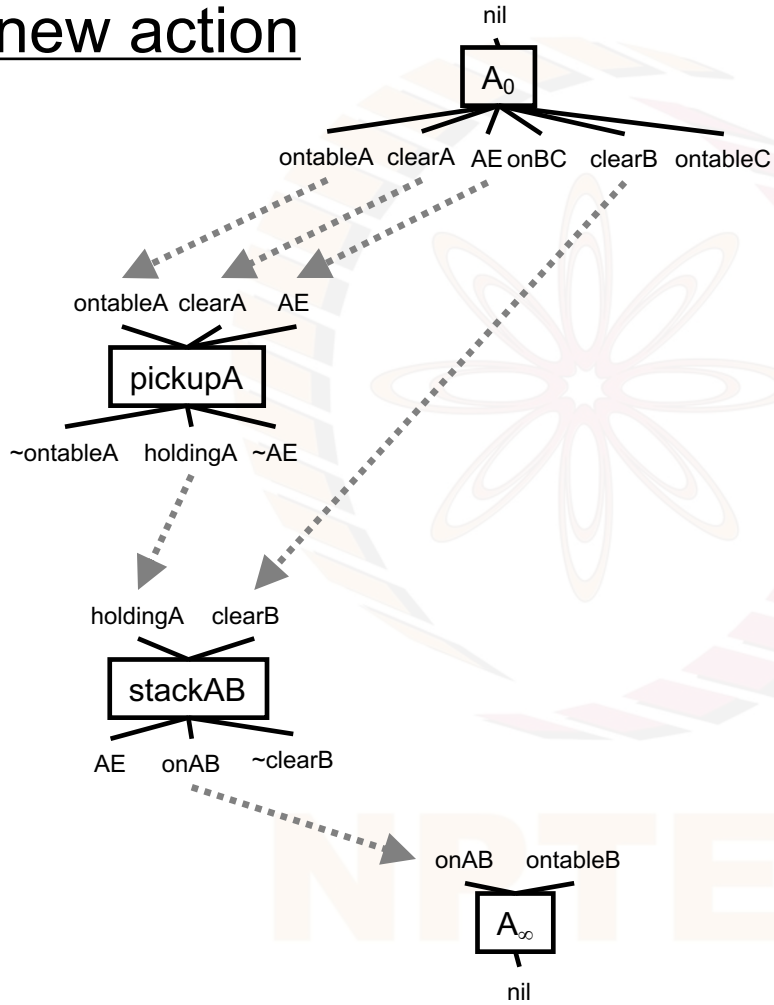
The new plan represents the set of all plans which have stack(A,B) in this position

Open goal : existing action



Action A_0 provides a causal link
for the open goal clear(B)

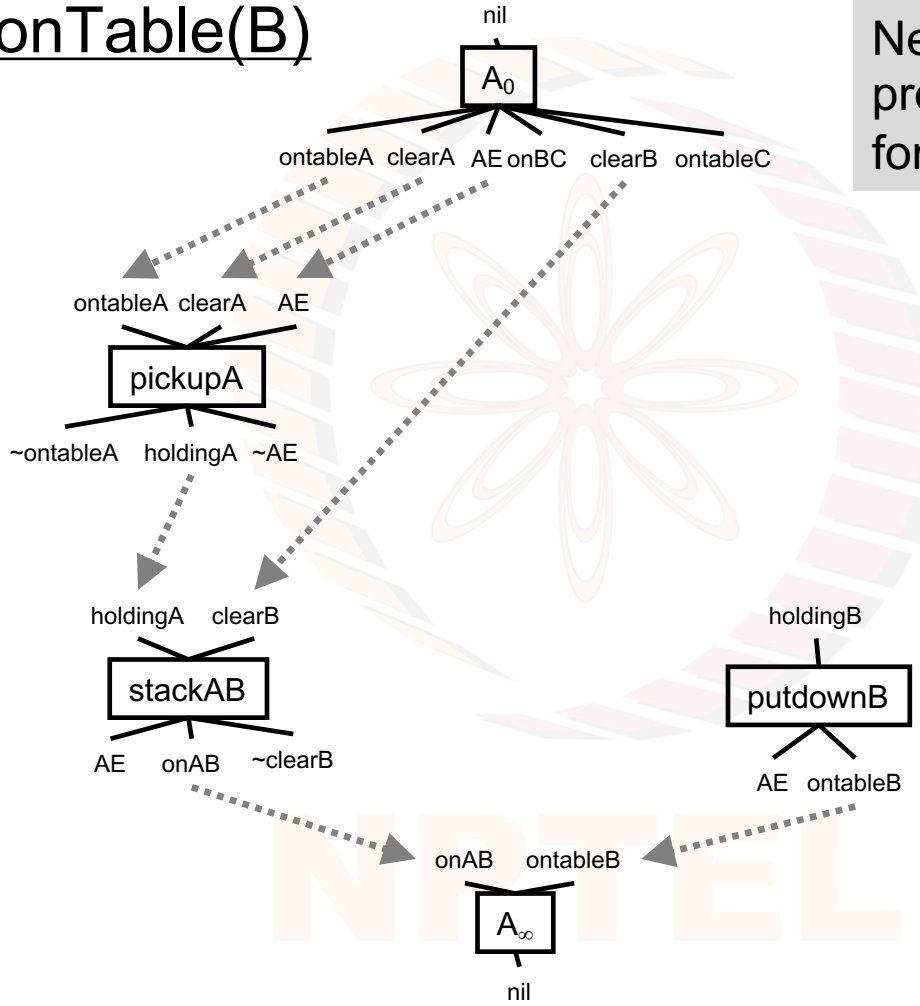
Open goal : new action



Action Pickup(A) provides a causal link
for the open goal holding(A)

The causal links for the preconditions of Pickup(A) are provided by A_0

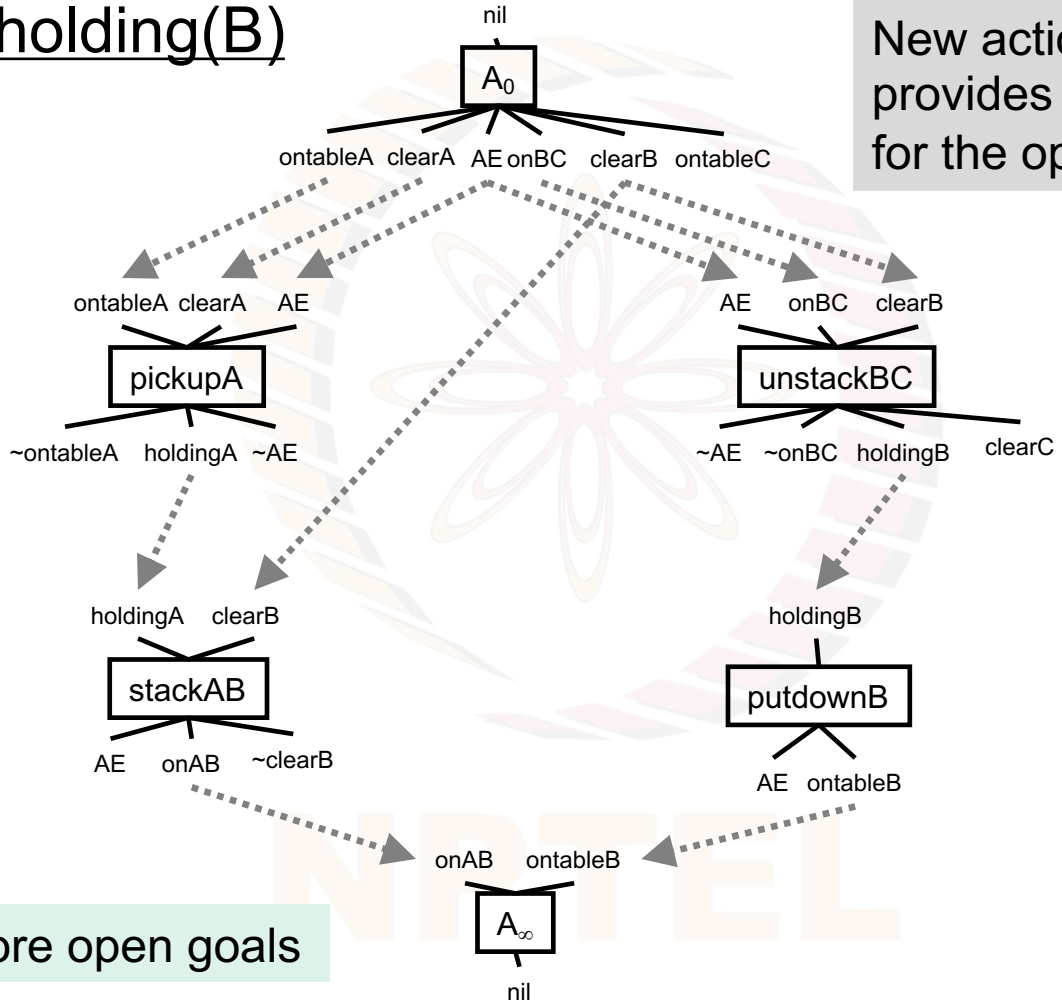
Open goal : onTable(B)



New action Putdown(B)
provides a causal link
for the open goal onTable(B)

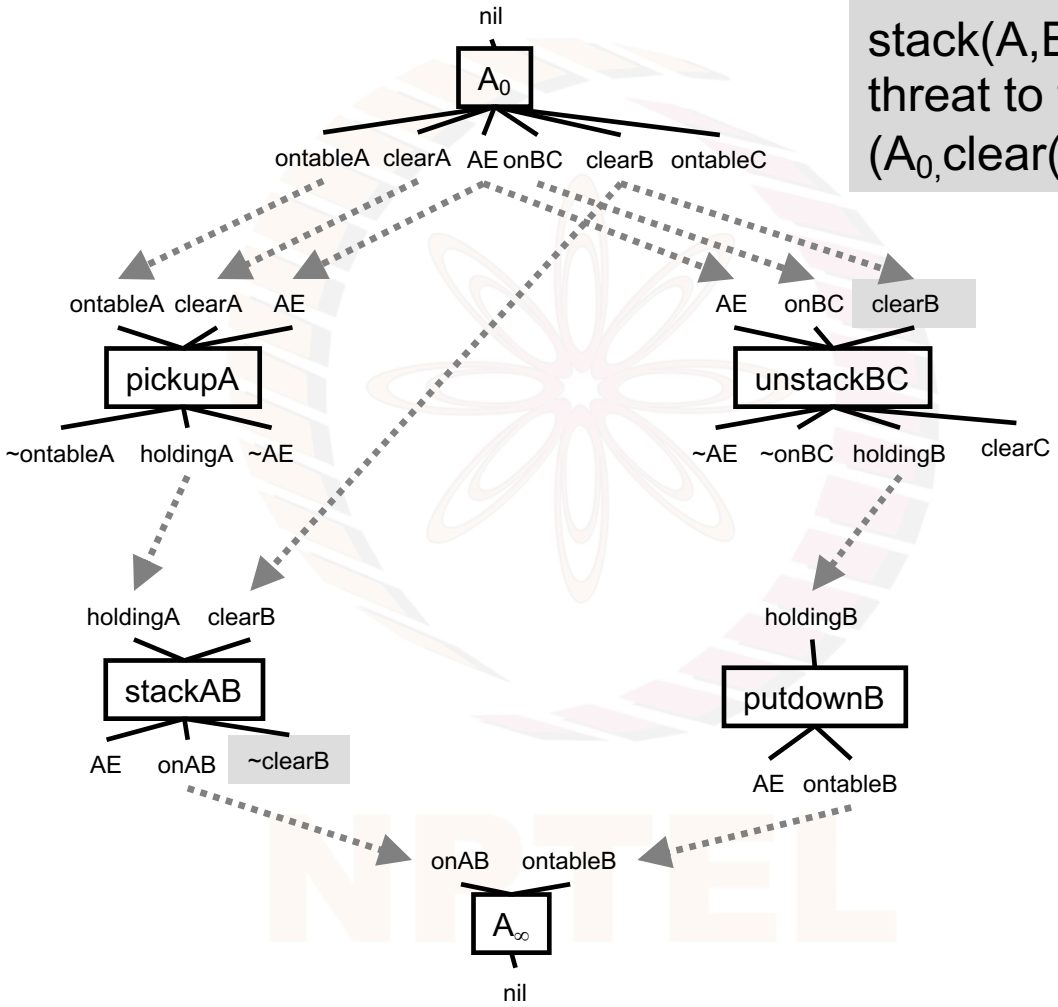
Open goal : holding(B)

New action Unstack(B,C)
provides a causal link
for the open goal holding(B)



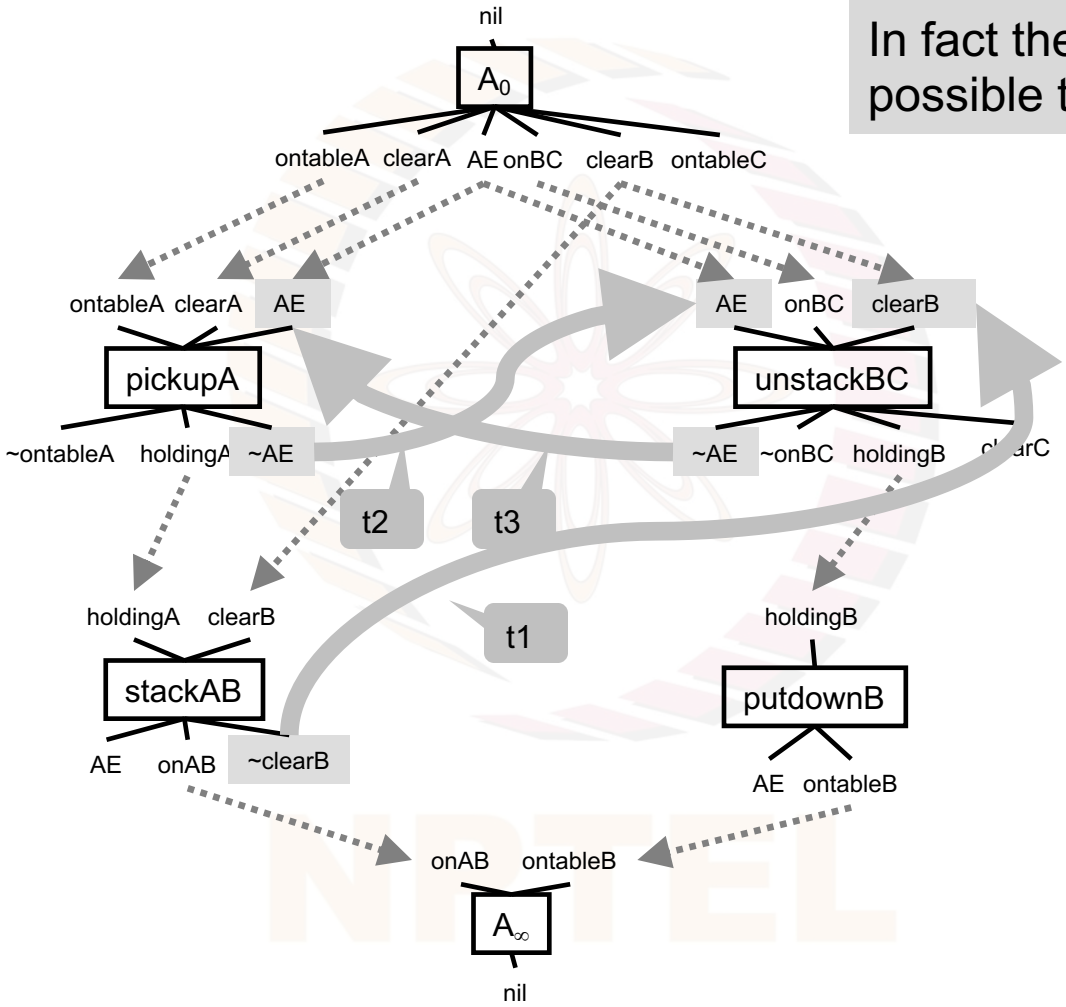
No more open goals

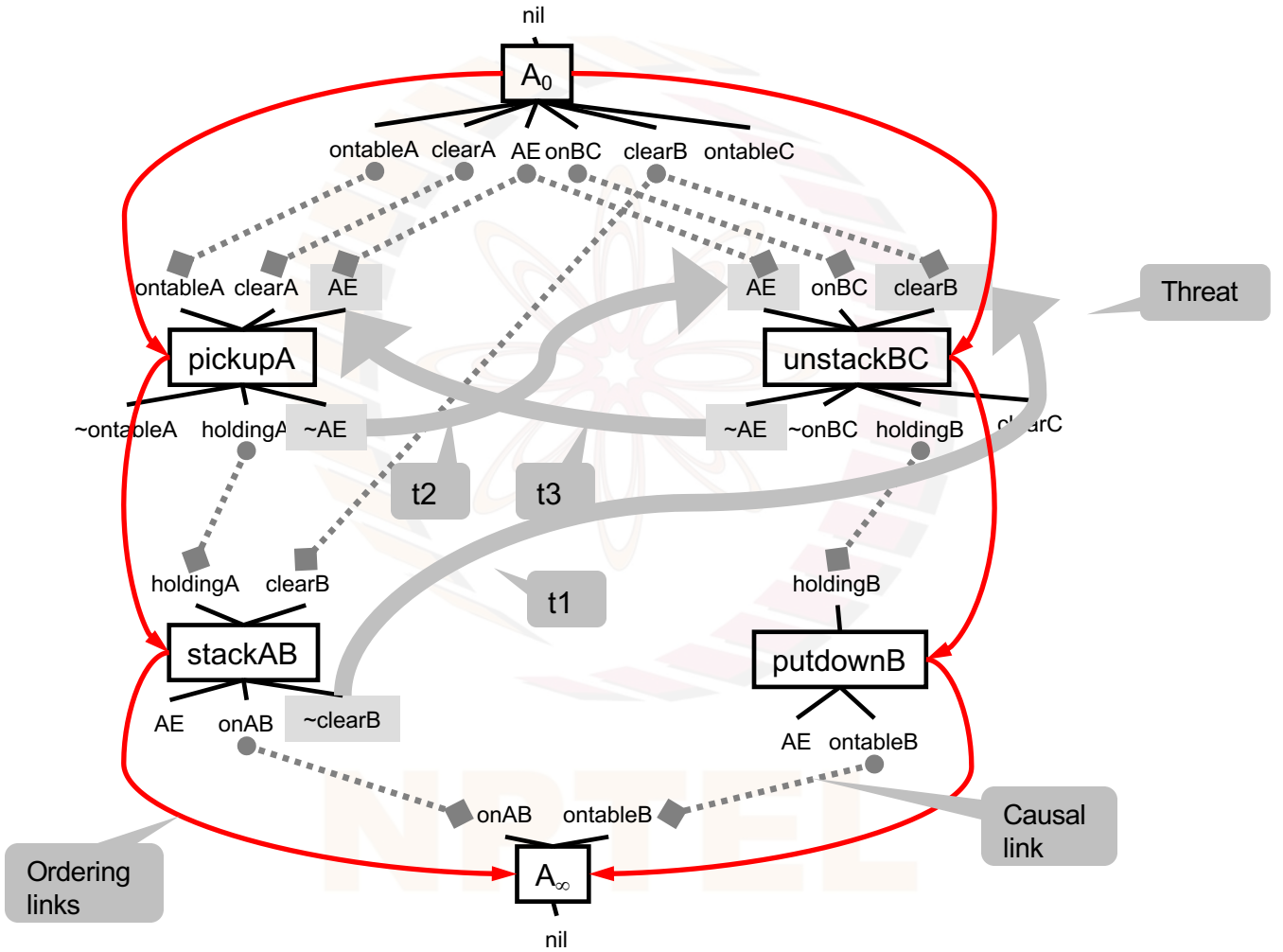
A threat

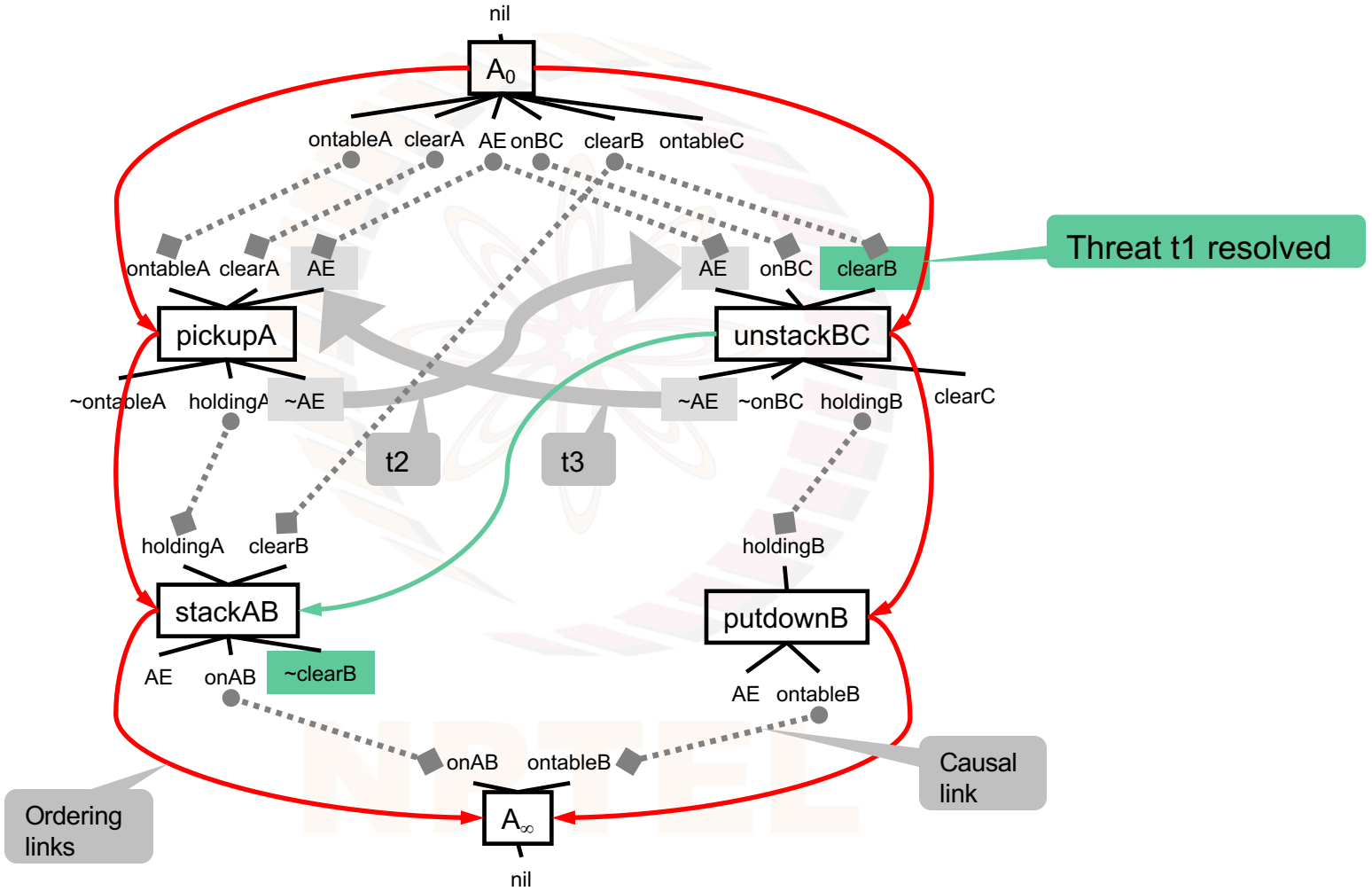


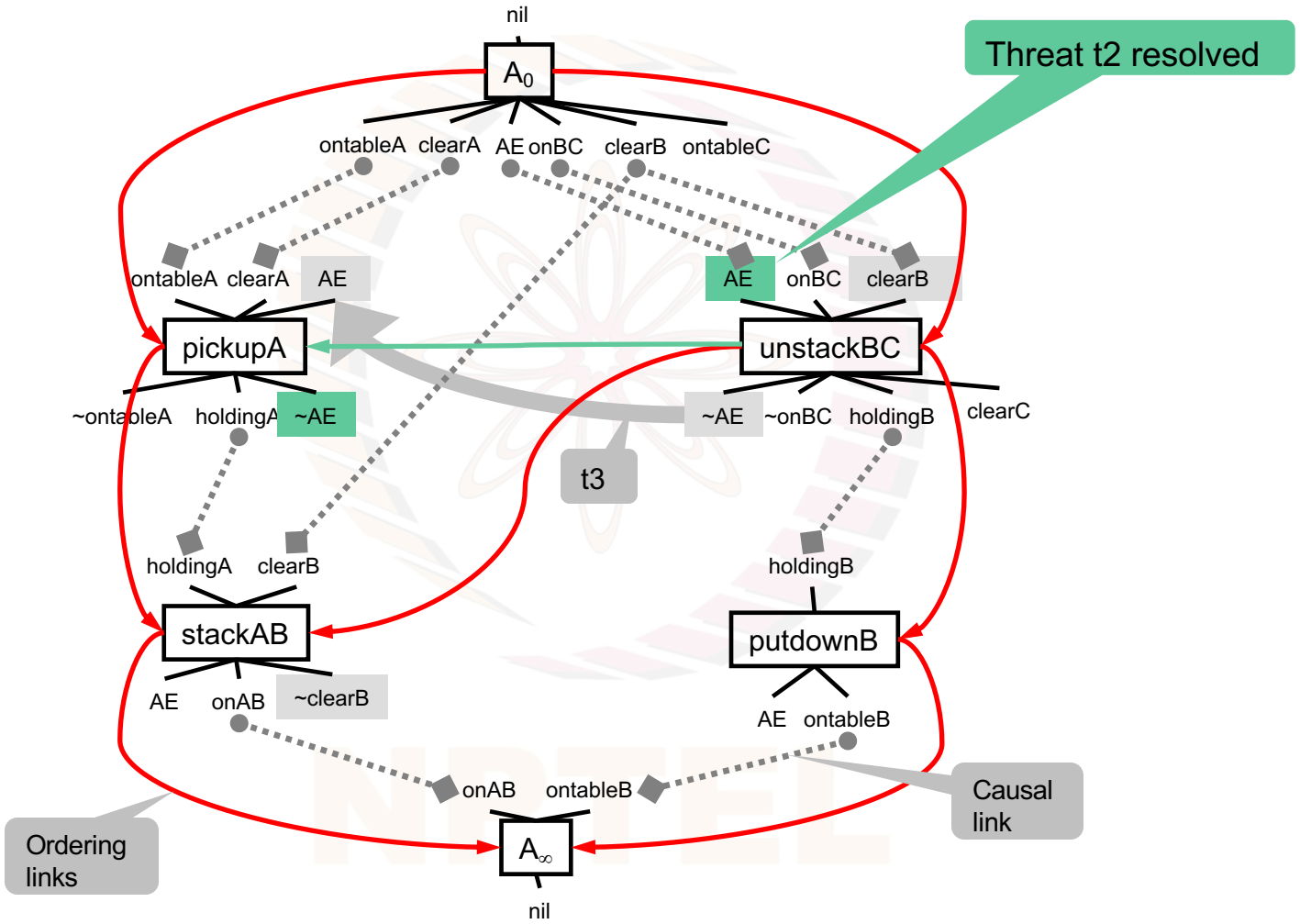
stack(A,B) is a possible threat to the causal link (A_0, clear(B), unstack(B,C))

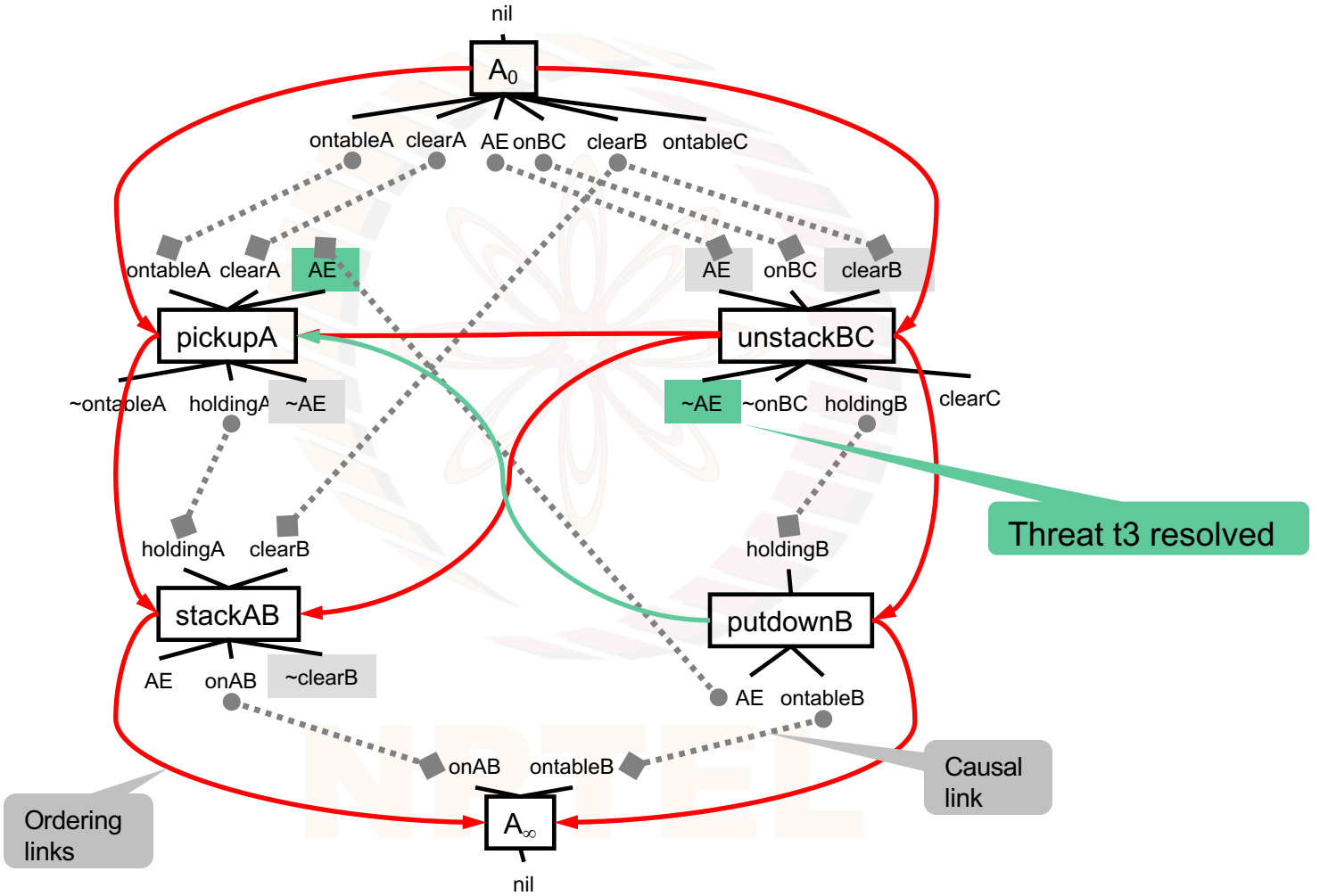
In fact there are three possible threats t1, t2 and t3




nil
A0
ontableA clearA AE onBC clearB ontableC
ontableA clearA AE
pickupA
~ontableA holdingA ~AE
AE onBC clearB
unstackBC
~AE ~onBC holdingB
Threat
t2
t3
t1
holdingA clearB
stackAB
AE onAB ~clearB
holdingB
putdownB
AE ontableB
onAB ontableB
A∞
nil
Ordering links
Causal link

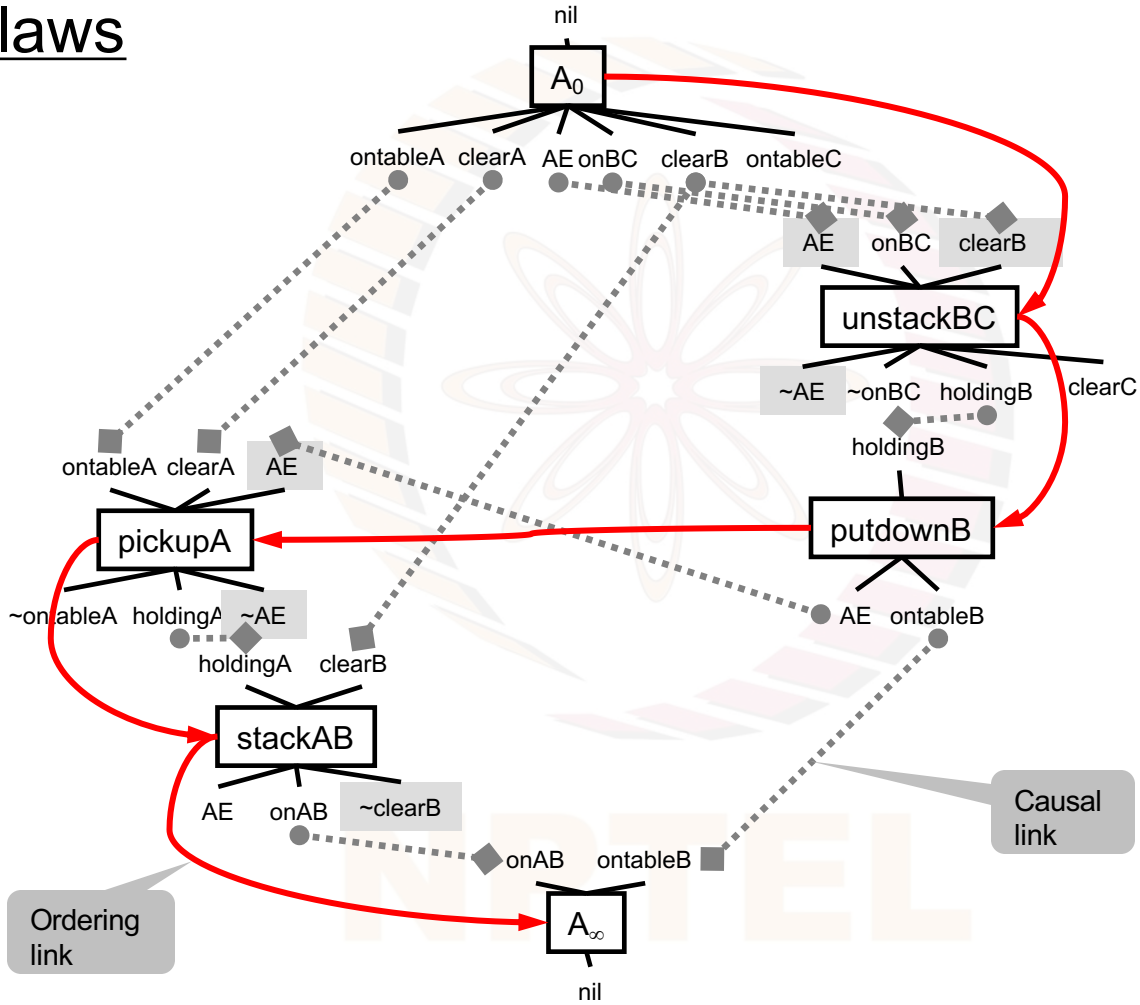

nil
A0
ontableA clearA AE onBC clearB ontableC
ontableA clearA AE
AE onBC clearB
Threat t1 resolved
pickupA
unstackBC
~ontableA holdingA ~AE
~AE ~onBC holdingB clearC
t2
t3
holdingA clearB
holdingB
stackAB
putdownB
AE onAB ~clearB
AE ontableB
onAB ontableB
Causal link
Ordering links
A∞
nil


nil
A0
ontableA clearA AE onBC clearB ontableC
Threat t2 resolved
ontableA clearA AE
pickupA
~ontableA holdingA ~AE
AE onBC clearB
unstackBC
~AE ~onBC holdingB clearC
t3
holdingA clearB
stackAB
AE onAB ~clearB
holdingB
putdownB
AE ontableB
onAB ontableB
A∞
nil
Ordering links
Causal link


nil
A0
ontableA clearA AE onBC clearB ontableC
ontableA clearA AE
AE onBC clearB
pickupA
unstackBC
~ontableA holdingA ~AE
~AE ~onBC holdingB clearC
holdingA clearB
holdingB
stackAB
putdownB
AE onAB ~clearB
AE ontableB
onAB ontableB
A∞
nil
Threat t3 resolved
Causal link
Ordering links

No more flaws

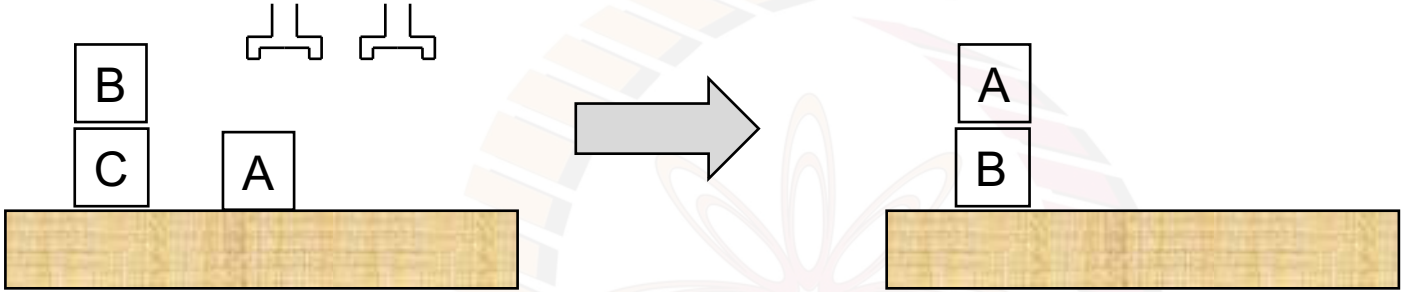
Linear plan
4 time steps



Versatile Multi Armed Robots
doing

Plan Space Planning
Partial Order Planning
Non-linear Planning
Least Commitment Planning

A two armed robot?



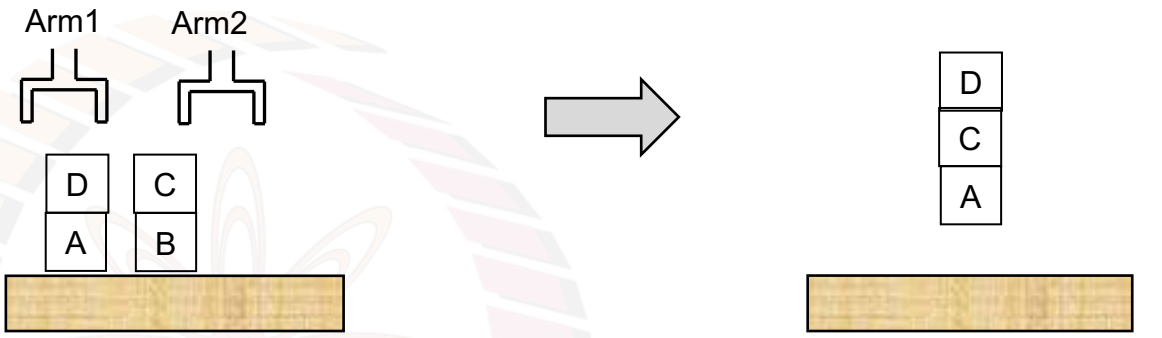
Given that we had a one armed robot, it was only possible to have a linear sequential plan, of 4 time steps

What if we had a two armed robot which could do actions in parallel? How long would the plan take to execute?

Exercise: Modify the STRIPS operators to cater to the domain with a two armed robot.

Will you solution extend easily to multiple arms?

A Two Armed Robot



Start

onTable(A), onTable(B),
on(D,A), on(C,B)
clear(D), clear(C), A1E, A2E

Goal

on(C,A), on(D,C)

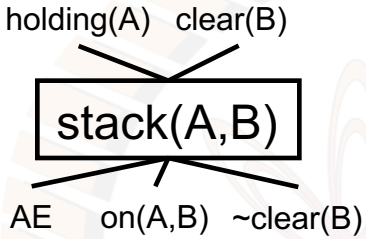
Exercise

Modify the STRIPS operators to work with two arms Arm1 and Arm2.

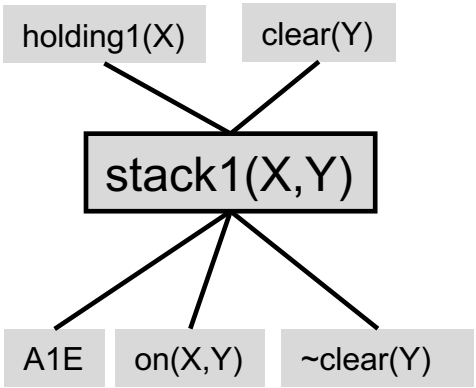
Show the plan when the Plan Space Planning algorithm terminates.

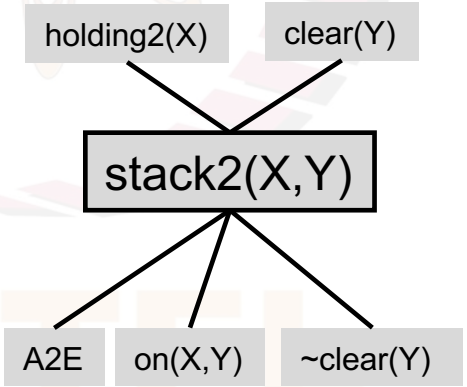
Modified STRIPS Planning Domain: Stack(X,Y)

One solution
Two operators for two arms



Assuming Pickup(X) and Unstack(X,Y) *do not* delete clear(X)



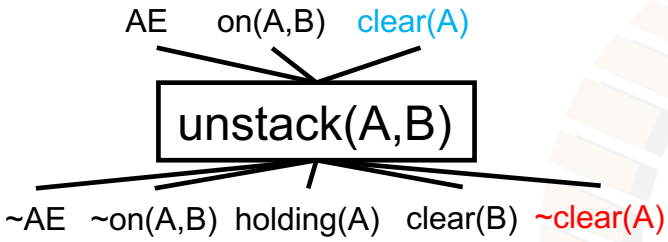


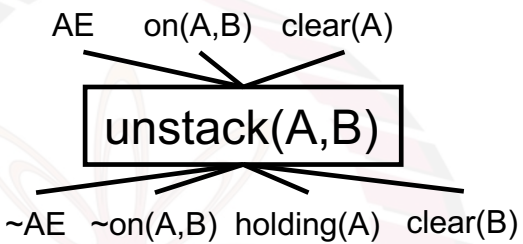
Predicates

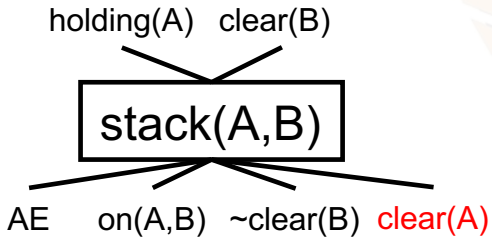
ontable(X)
on(X,Y)
clear(X)
holding1(X)
holding2(X)
A1E
A2E

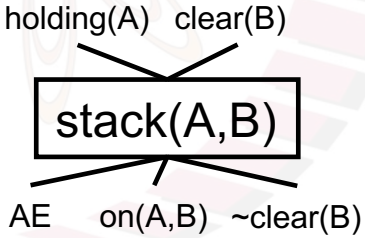
Similarly for other operators. What if the robot had multiple arms?

To delete or not to delete .... clear(X)









Predicates

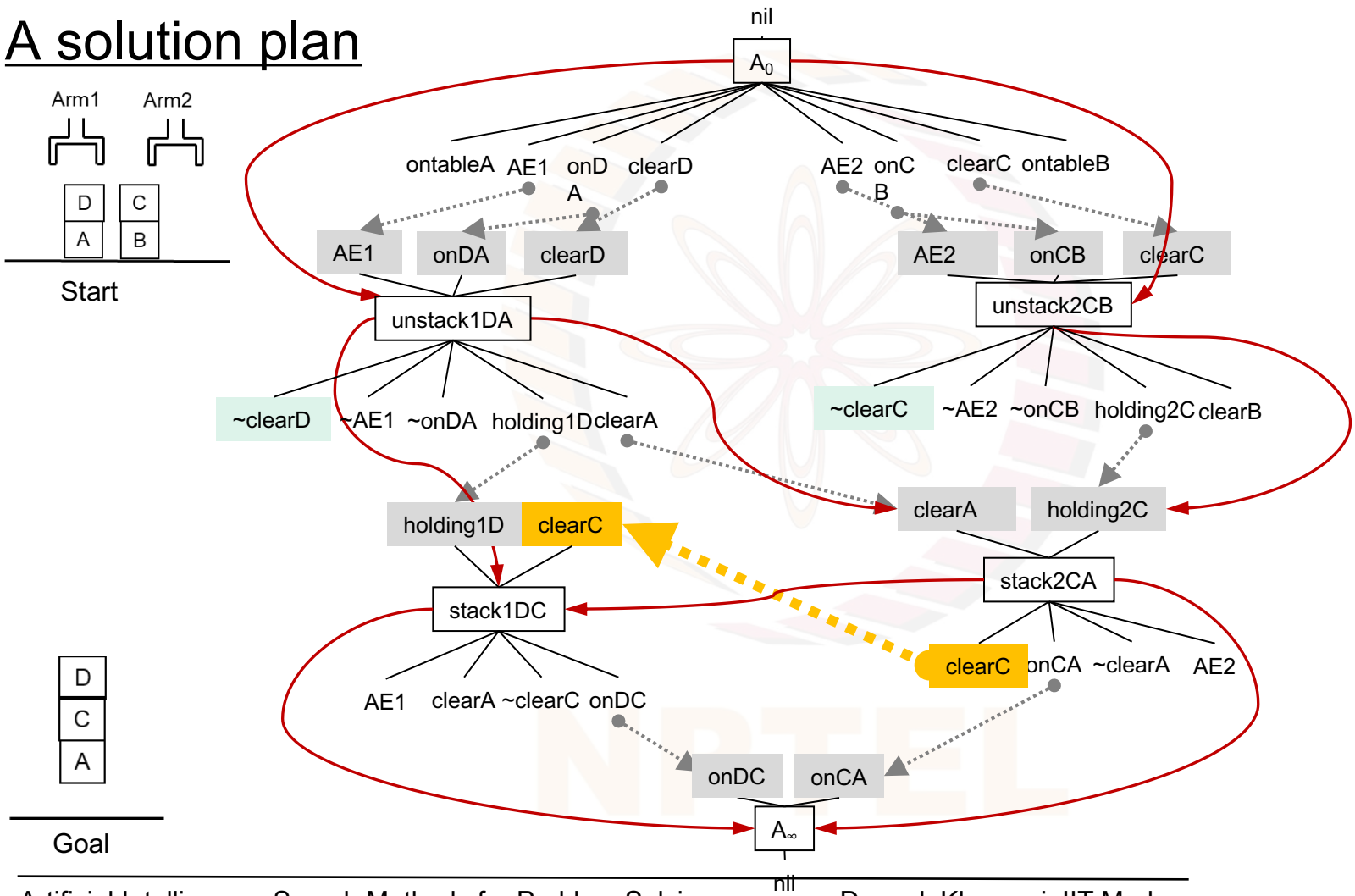
ontable(X)
on(X,Y)
clear(X)
holding(X)
AE

An alternative... after unstack(A,B) should clear(A) be deleted?

The operators presented so far.

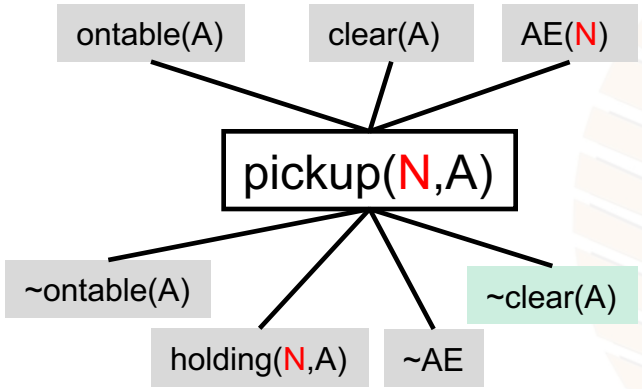
In this one armed robot domain
the *only* thing that can happen is stack(A,?X) or putdown(A)

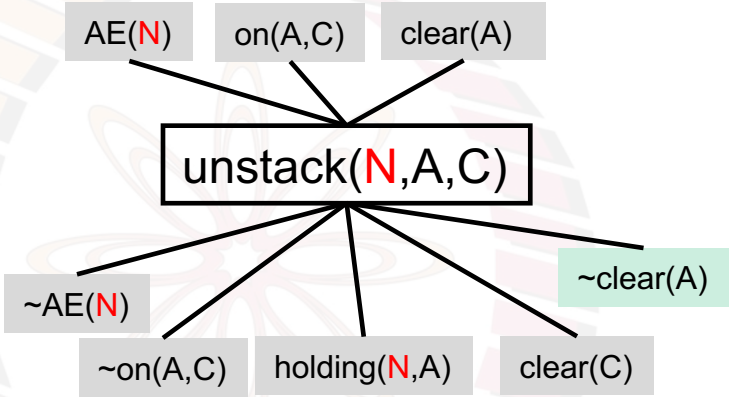
A solution plan

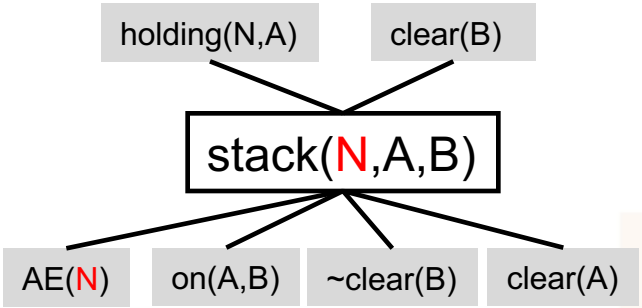


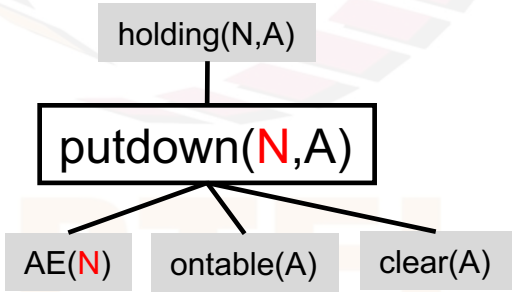
Multi-armed robot domain 1

Since A is not clear nothing can be stacked on A





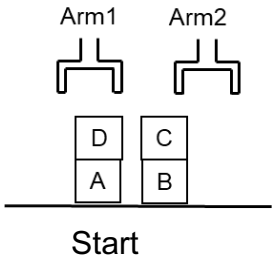




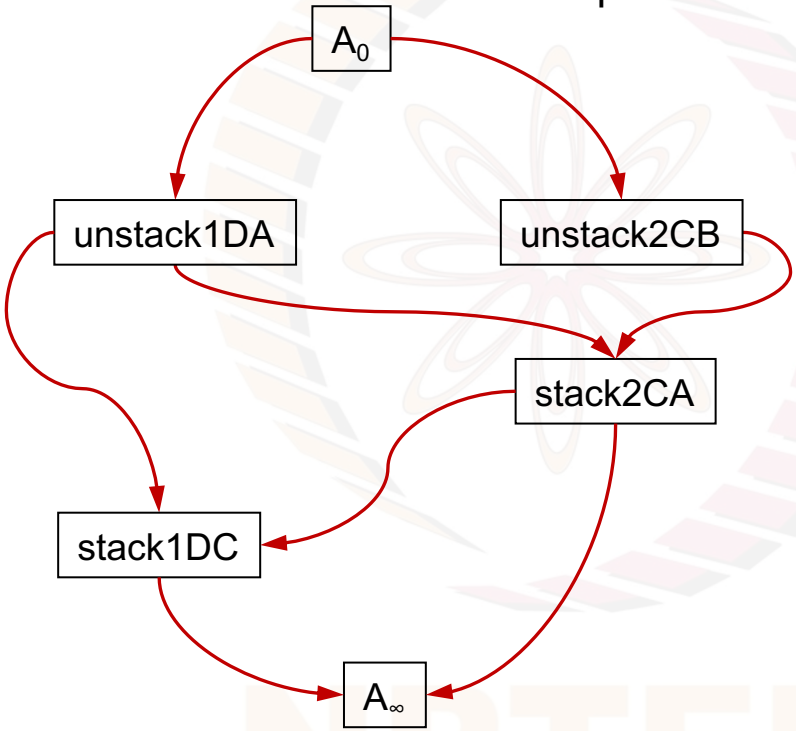
Predicates

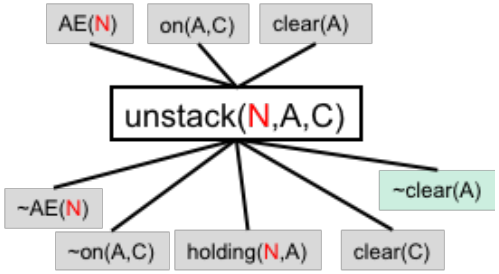
ontable(X)
on(X,Y)
clear(X)
holding(N,X)
AE(N)

Domain description 1



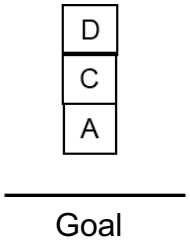
both unstack actions can be done in parallel





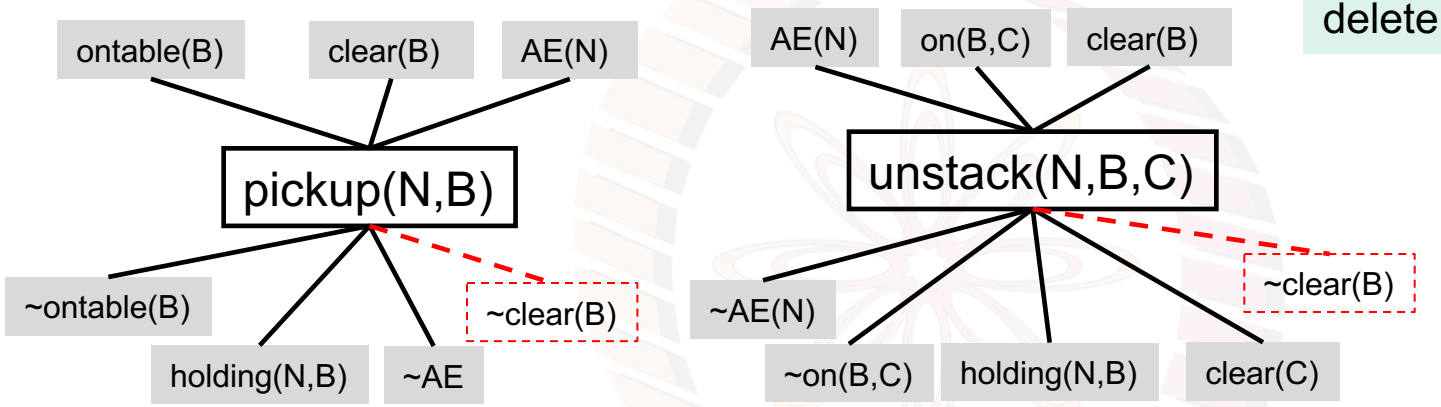
since clear(C) is *not true*, *only* stack2(C,A) can be done

stack1(D,C) has to wait for clear(C) to become true by the stack2(C,A) action



Multi-armed robot domain 2

Assuming Pickup(N,X) and Unstack(N,X,Y) do not delete clear(X)

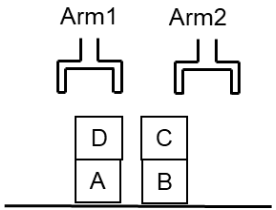


Since holding(N,B) and clear(B) are *both* true
the action *stack(M,A,B) is applicable*

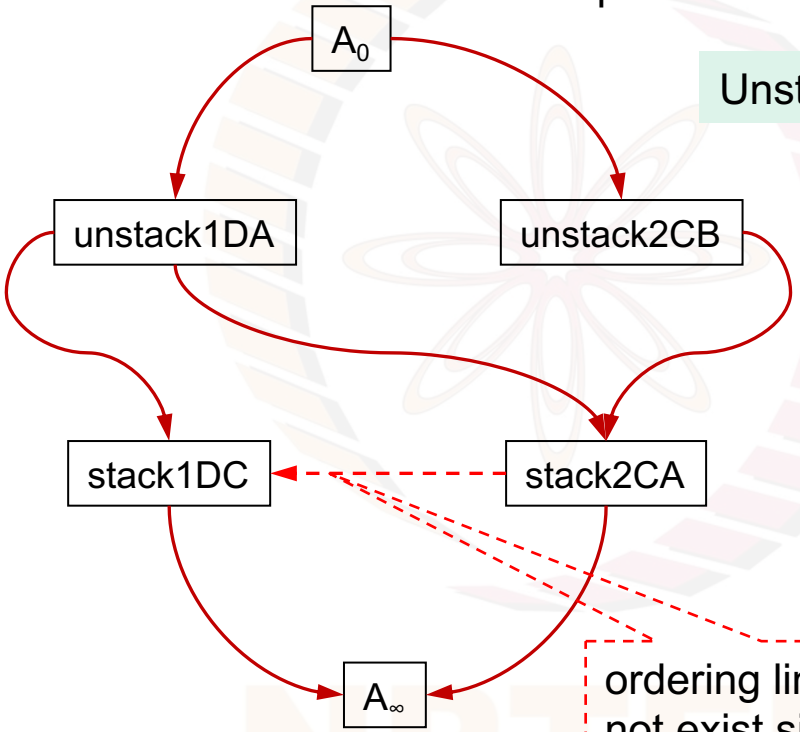
- robot can *hold a stack of blocks*!
- also putdown(N,B) and stack(M,A,B) *can be concurrent*!
- *many arms can create a stack in one time step*!

Exercise: Modify the domain so that an arm can hold *only* 2 blocks

Domain description 2



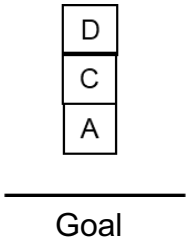
both unstack actions can be done in parallel



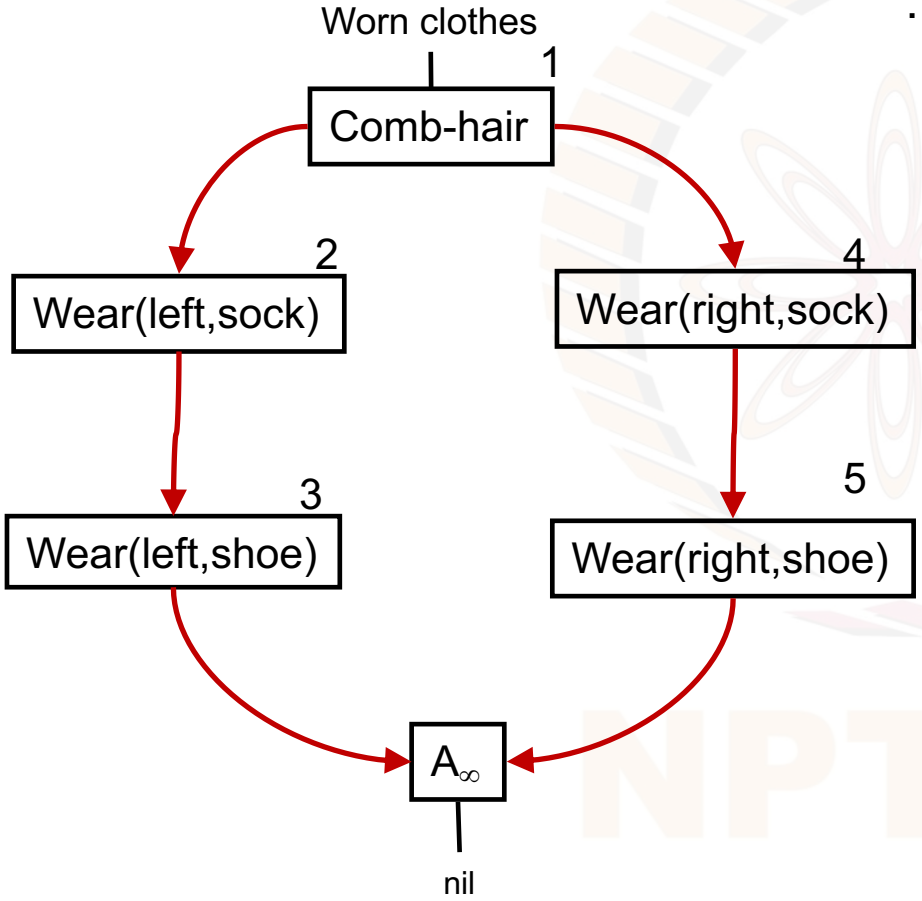
Unstack(N,X,Y) does not delete clear(X)

since clear(C) remains *true*, *both* stack2(C,A) and stack1(D,C) *can be* done in parallel

ordering link does not exist since there is no causal link



Executing a partial plan



Actions 2 and 4 can be done in parallel
... as can actions 3 and 5

Any topological sort of the *partial order* is a valid linear plan as well

1-2-3-4-5
1-2-4-3-5
1-2-4-5-3
1-4-2-3-5
1-4-5-2-3
1-4-2-5-3

Action selection, action scheduling

A plan is a set of actions designed to transform the given state into the desired or goal state when executed.

- so far we *had* said a plan is a sequence of actions
- $\pi = \langle a_1, a_2, ..., a_n \rangle$
- but actions can now happen in parallel
- and actions could have durations

- in FSSP the first action selected is the first one scheduled
- in BSSP the first action selected is the last one scheduled
- in GSP the first action proposed is the last one to be scheduled

- In PSP there is the possibility of
  - *separating* the task of *selecting* actions
  - from *scheduling* them
  - and devising plans that can execute in parallel

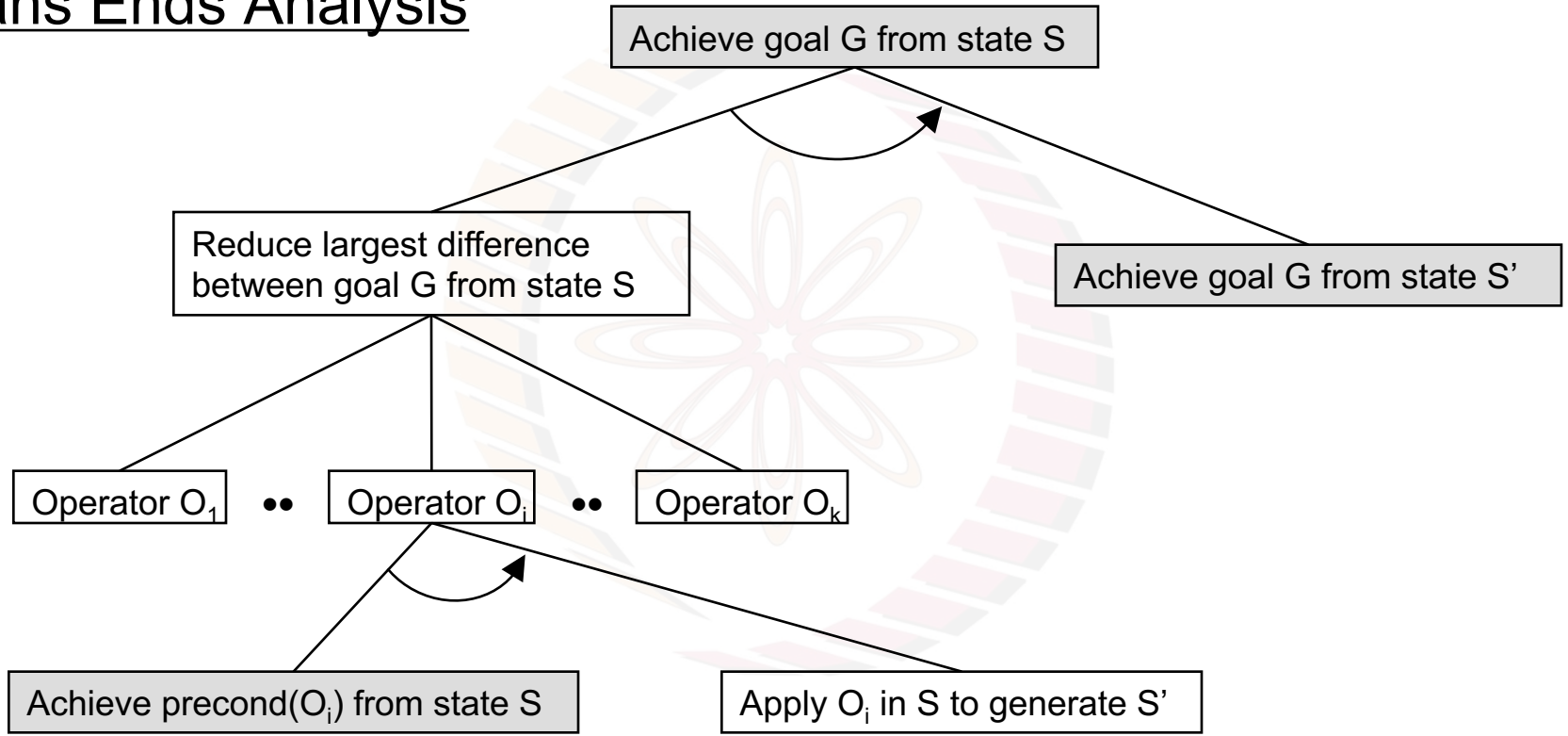
Means Ends Analysis

In their seminal work on *Human Problem Solving*, Newell and Simon proposed a general purpose strategy for problem solving, which they call the *General Problem Solver* (GPS). GPS encapsulated the heuristic approach which they called *Means Ends Analysis*.

- compare the current state with the desired state, and
- list the *differences* between them
- evaluate the differences in terms of magnitude (in some way)
- consult an *operator-difference table*
- reduce *largest* (most important) *differences first*

- the *differences* characterize the *ends* that need to be achieved
- the *operators* define the *means* of achieving those ends.

Means Ends Analysis



The MEA strategy generates an AND/OR tree by replicating the this structure below the shaded nodes when a recursive call is made.

Travel

Let us say you want to travel from IIT Madras to the Parashar lake in Himachal

- we measure the differences in terms of distances
- in the light of operators that can reduce such differences
- the operator difference table could look like the following

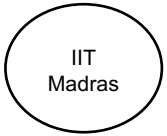
Modes of transport

Distances	Airplane	Train	Car	Taxi	Bus	Walk
More than 5000 km	yes					
100 km to 5000 km	yes	yes	yes			
1 km to 100 km		yes	yes	yes	yes	
Less than 2 km			yes		yes	yes

At IIT Madras, not at Parashar

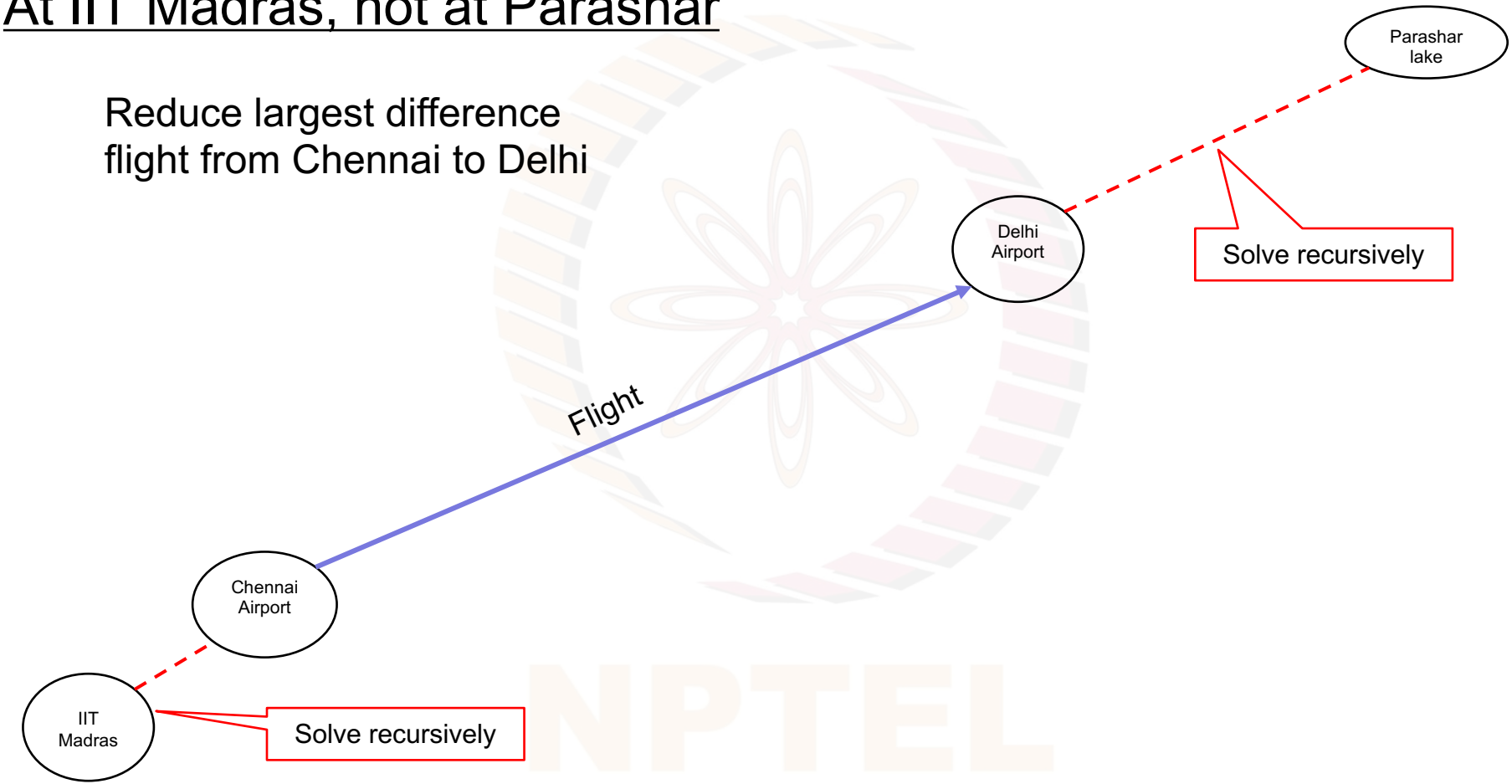
Differences = geographical distances

Means – Flights, Train, Bus, Taxi, Walk



At IIT Madras, not at Parashar

Reduce largest difference
flight from Chennai to Delhi

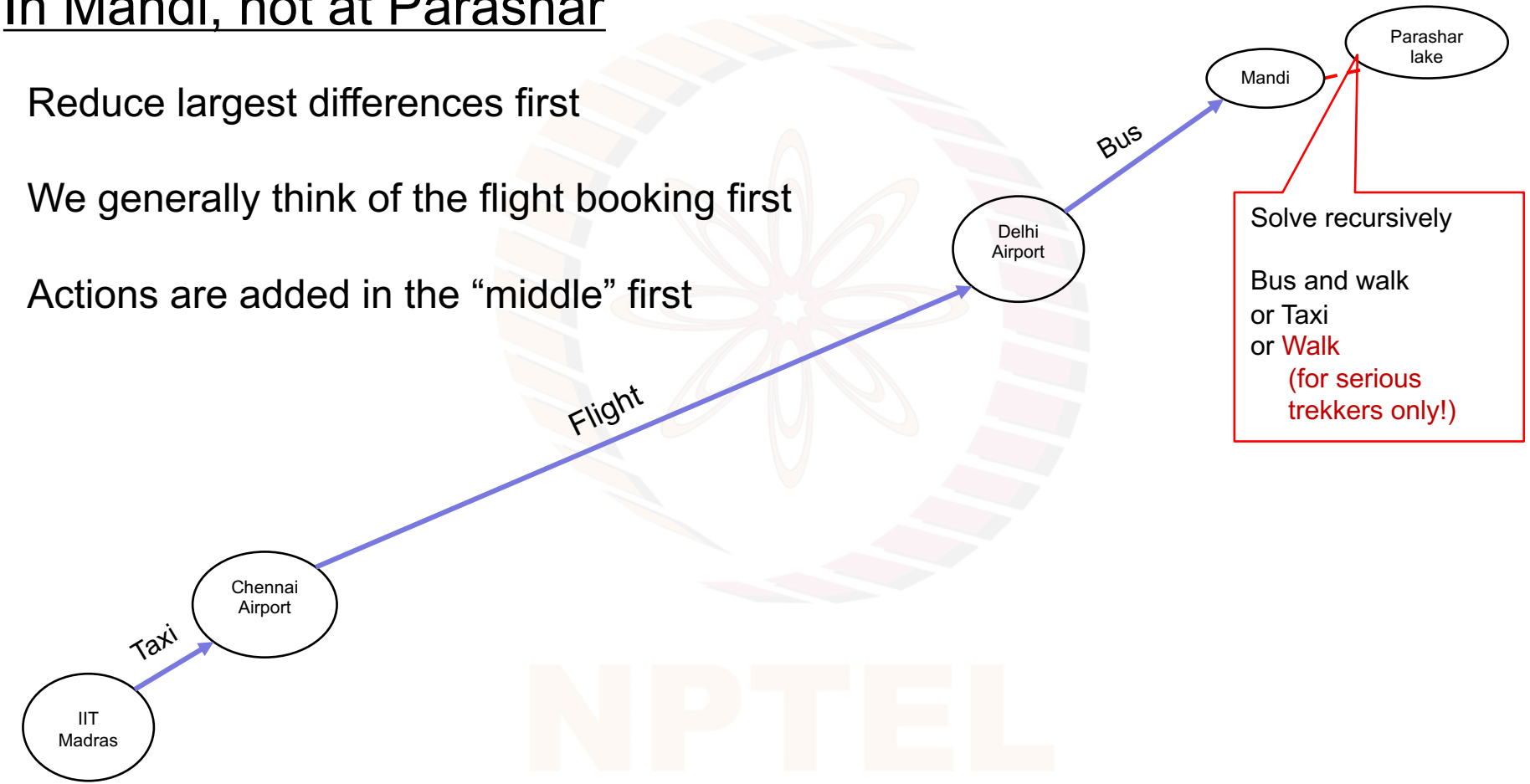


In Mandi, not at Parashar

Reduce largest differences first

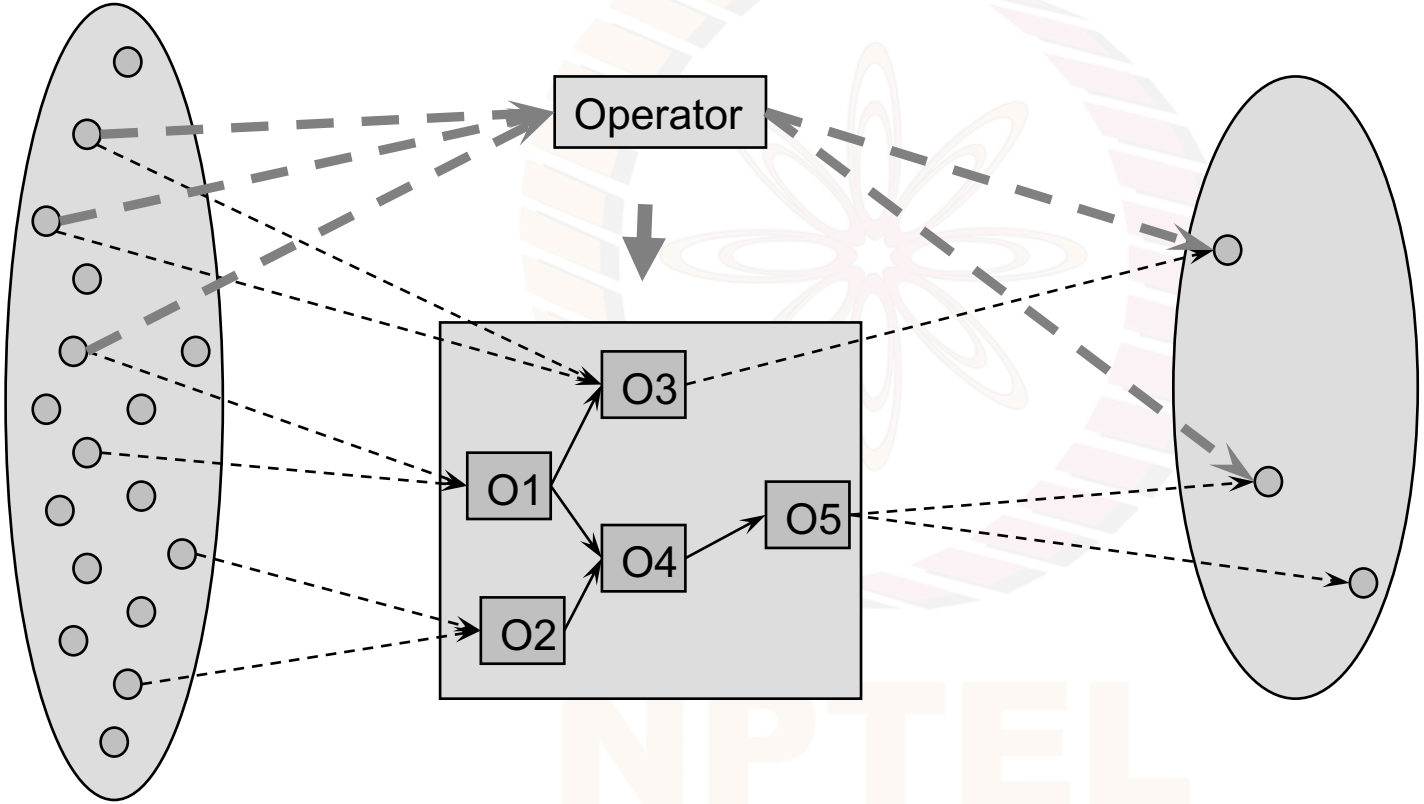
We generally think of the flight booking first

Actions are added in the “middle” first



Hierarchical Planning

High level operators can be refined to networks of lower level operators



Next

Algorithm Graphplan
An entirely new approach

Algorithm Graphplan

The algorithm *Graphplan* presented by Avrim Blum and Merrick Furst (1995) takes a very different view of the planning problem.

Instead of searching for a solution either in the state space or the plan space

- it first constructs a structure called a *planning graph* that potentially captures all possible solutions and
- then proceeds to search for a solution in the planning graph

Starting with Graphplan a variety of new planning algorithms burst forth...

These algorithms increased the lengths of plans
that could be constructed by an order of magnitude –
from tens of actions to hundreds of actions

Two stage planning

Graphplan pioneered the approach to two stage planning

- construct a planning graph and search for solution in the planning graph

Satplan - Henry Kautz, Bart Selman and David McAllester (1992)

- convert the planning to a satisfiability problem
- use an off-the shelf solver to solve the SAT problem

CPlan - Peter van Beek and Xinguang Chen (1999)

- convert the planning to a constraint satisfaction problem
- use an off-the shelf solver to solve the CSP

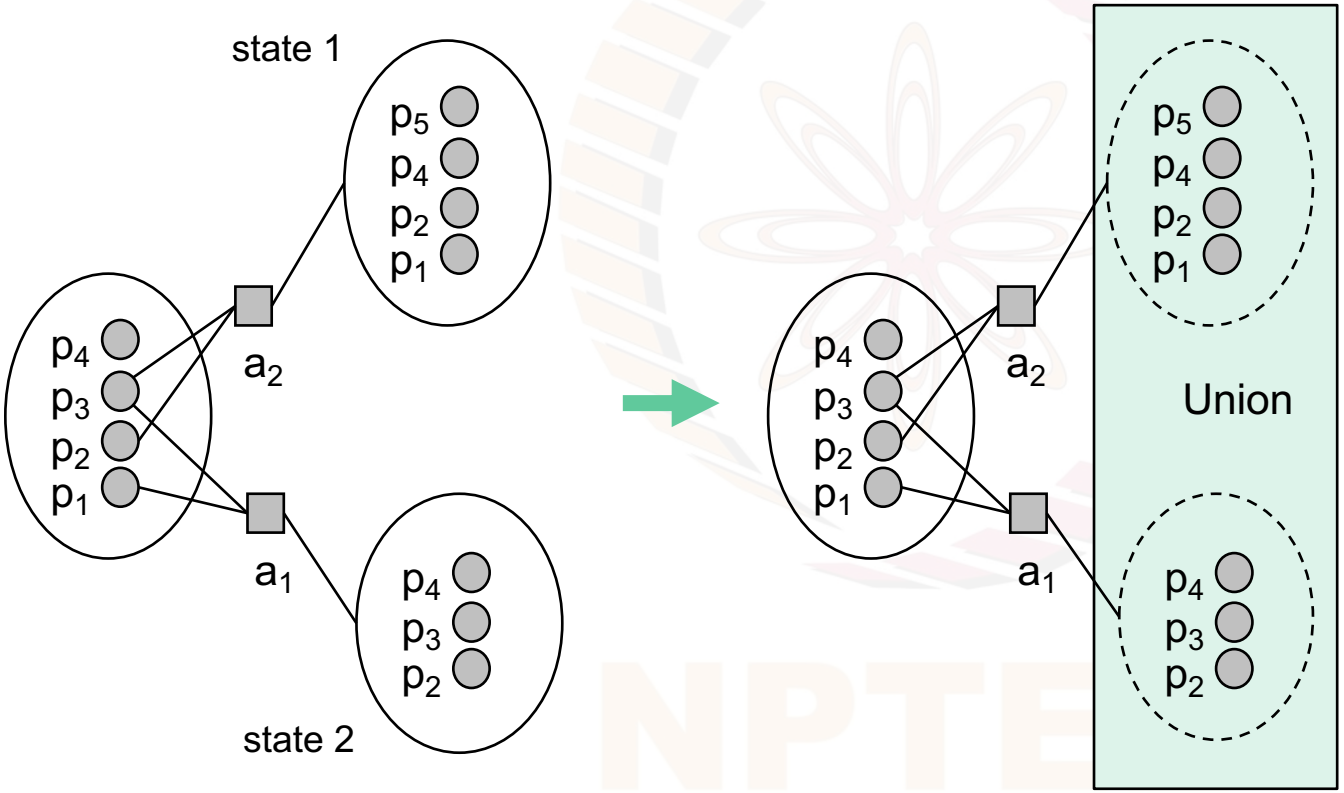
Another direction of research was Heuristic Search planners

- domain independent heuristics to guide state space search

We will conclude our study of planning with Graphplan

State Space vs. Planning Graph

the *proposition layer* constitutes of the *union* of propositions in all states at a given depth



The proposition layer in the planning graph

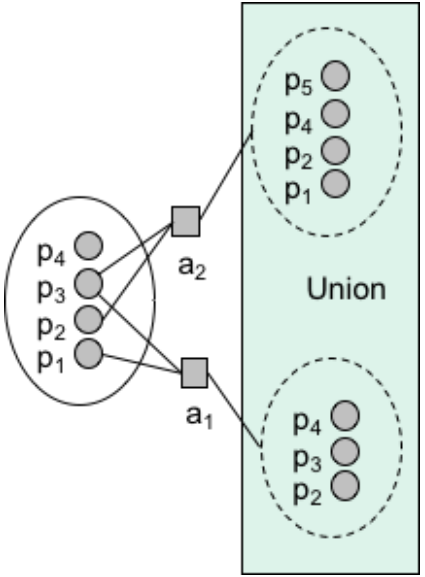
The figure shows the basic difference between the state space for a planning problem and the corresponding planning graph.

State space search generates a successor candidate *state*, which will be a starting point for further exploration.

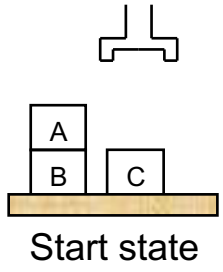
The planning graph is a structure, as shown on the right, which *merges* the states produced by the *different actions* that are applicable.

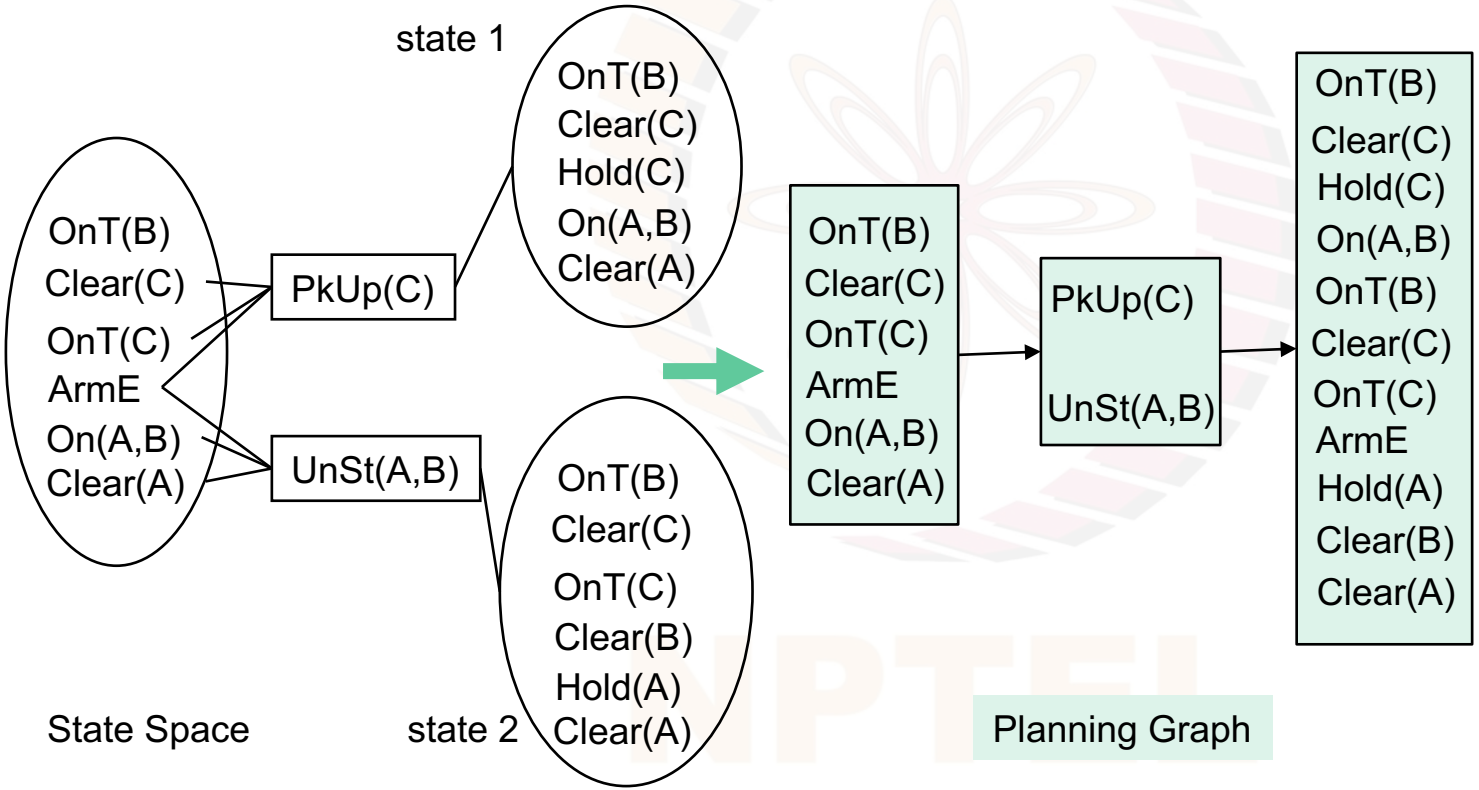
The resulting *set of propositions forms a layer*, as does the set of *actions* that resulted in this layer (see next slide).

The *planning graph* is a *layered graph* made up of such *alternating* sets of *action* layers and *proposition* layers.



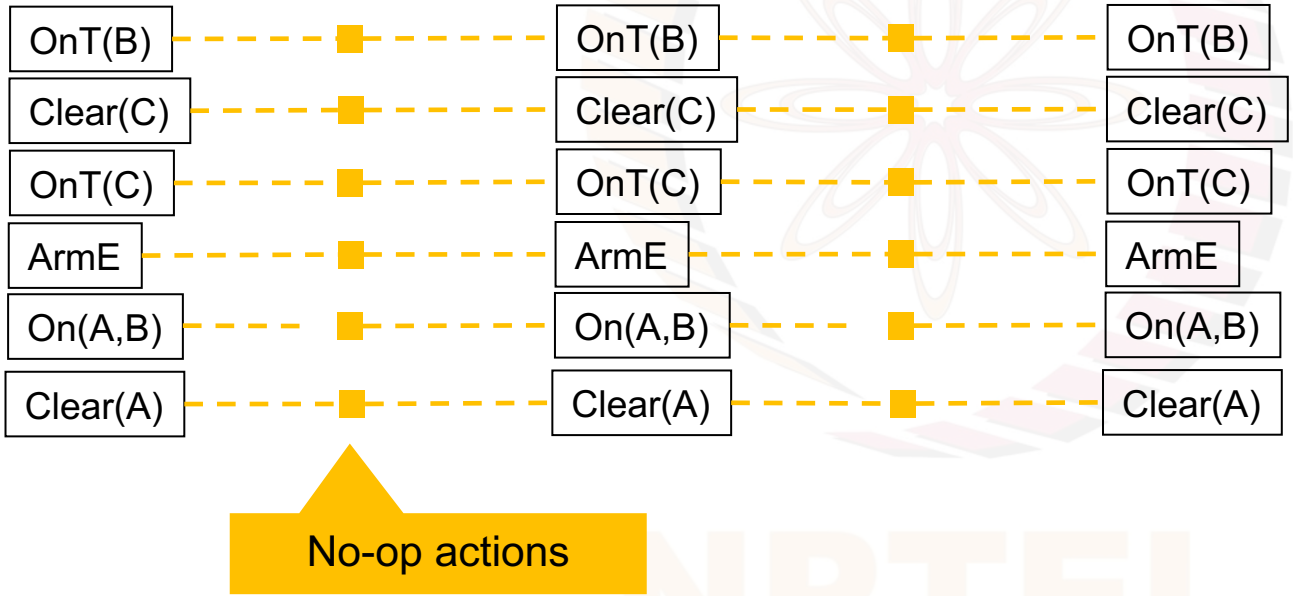
Action layer: *set* of all *individually applicable* actions

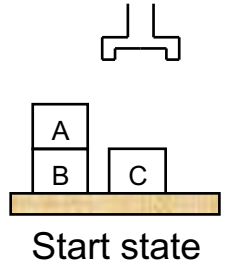




The *no-op* action

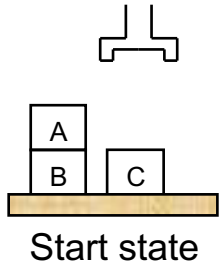
The *no-op* action is always applicable.
It has one *precondition P* and one *positive effect P*.
Thus *every* proposition is *replicated* in *every proposition layer*.

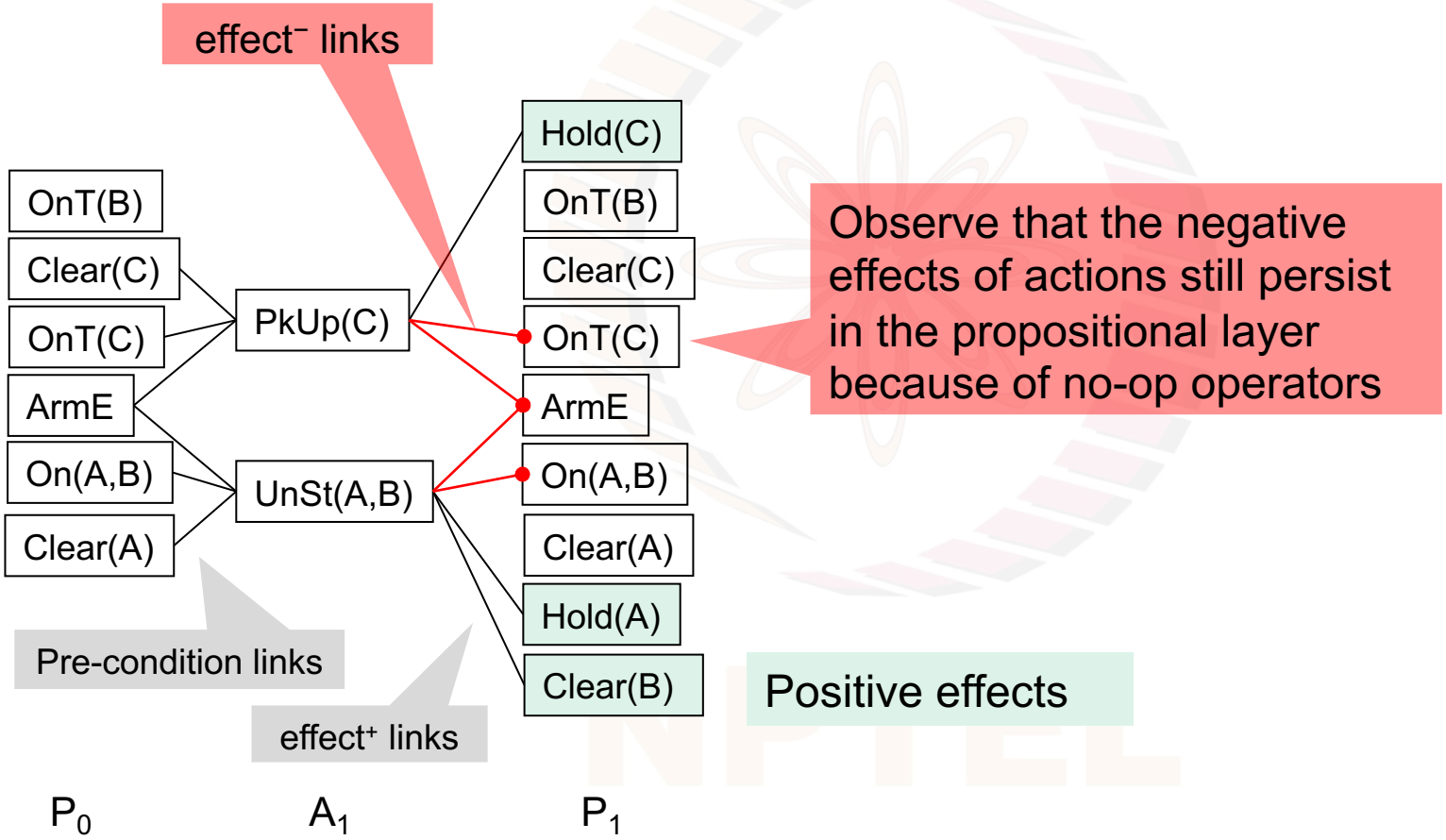




We will assume the *no-op* actions to be implicit in our figures.

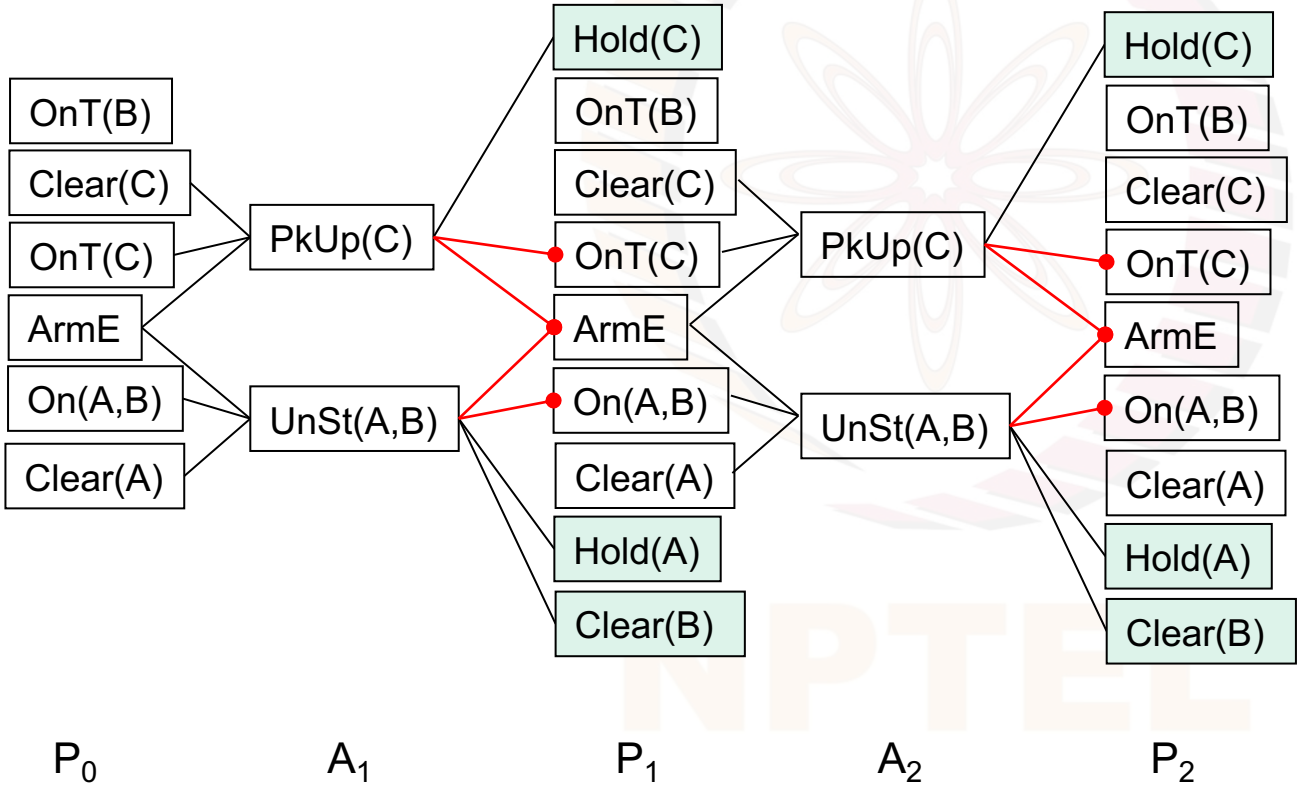
Action layer: set of all *individually applicable* actions

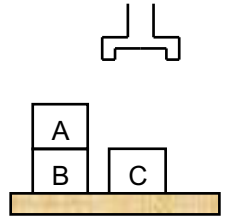




Action layer: set of all *individually applicable* actions

Because of *no-op* actions the actions and propositions in the $(i-1)^{th}$ layer get replicated in the i^{th} layer



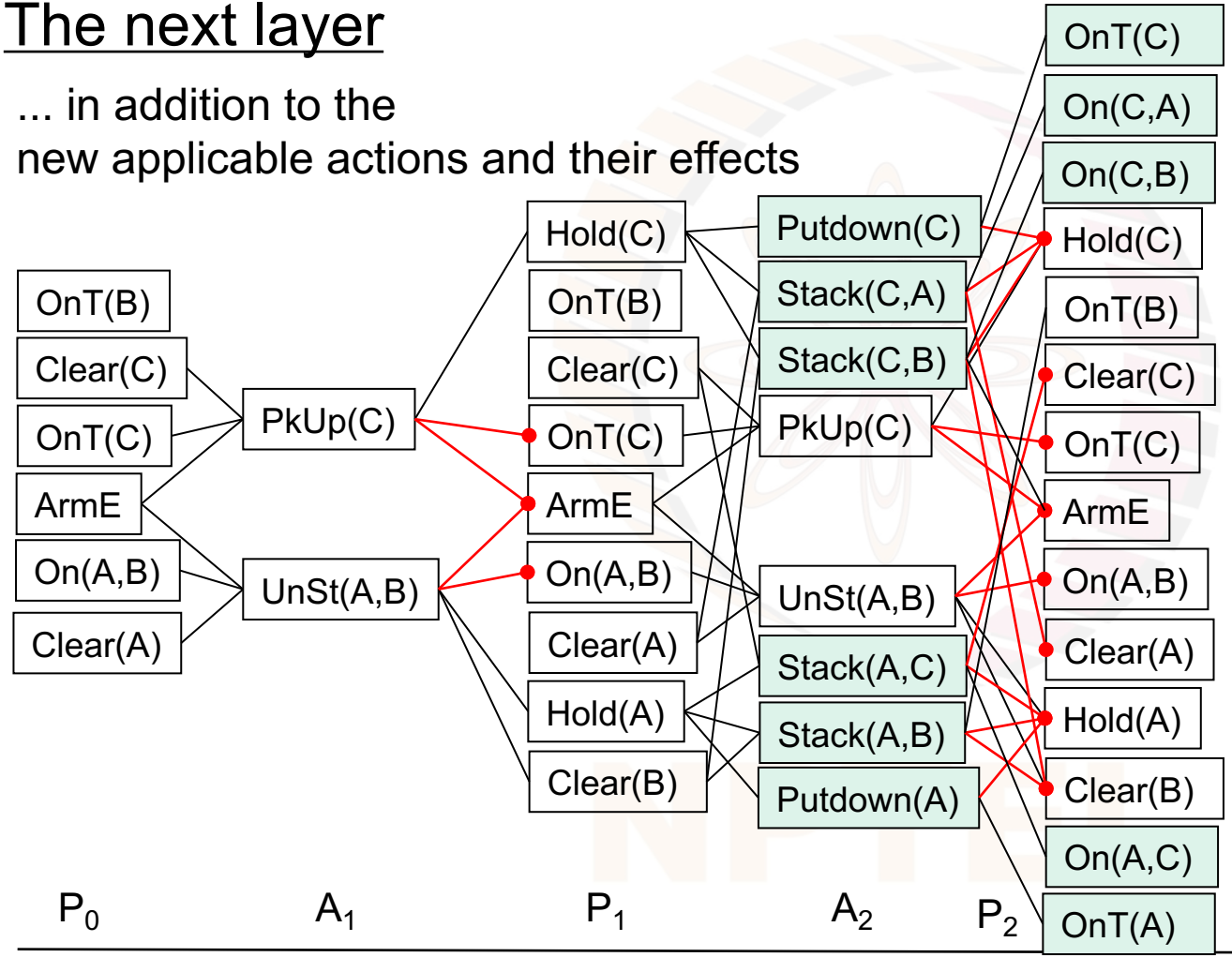


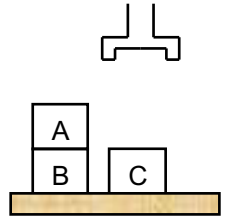
Start state

As a result the action and the proposition layers grow monotonically

The next layer

... in addition to the
new applicable actions and their effects





Start state

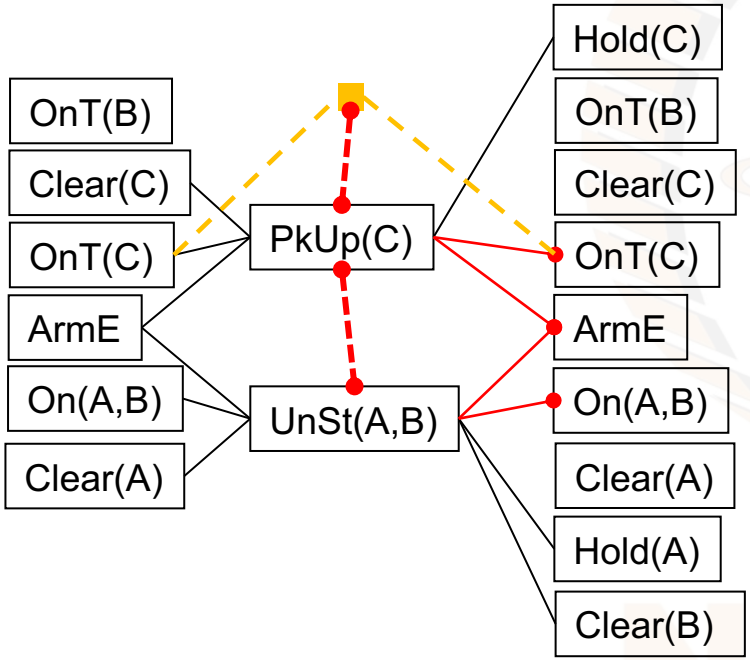
Note: Not all effect links have been drawn here!

Mutual Exclusion

Two actions $a \in A_i$ and $b \in A_i$ are *mutex* if one of the following holds,

1. *Competing needs*: There exist propositions $p_a \in pre(a)$ and $p_b \in pre(b)$ such that p_a and p_b are mutex (in the preceding layer).
2. *Inconsistent effects:* There exists a proposition p such that $p \in effects^+(a)$ and $p \in effects^-(b)$ or *vice versa.* The semantics of these actions in parallel are not defined. And if they are linearized then the outcome will depend upon the order.
3. *Interference:* There exists a proposition p such that $p \in pre(a)$ and $p \in effects^-(b)$ or *vice versa.* Then only one linear ordering of the two actions would be feasible.
4. There exists a proposition p such that $p \in pre(a)$ and $p \in effects^-(a)$ *and also* $p \in pre(b)$ and $p \in effects^-(b)$. That is the proposition is consumed by each action, and hence only one of them can be executed.

Mutex actions



Pickup(C) and *Unstack(C)* are mutex because

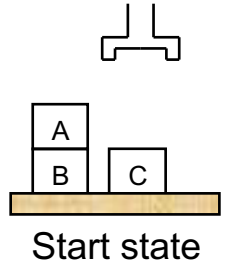
(4) both consume *ArmE*

Pickup(C) is also mutex with *onTable(C)-no-op-onTable(C)* because (2)

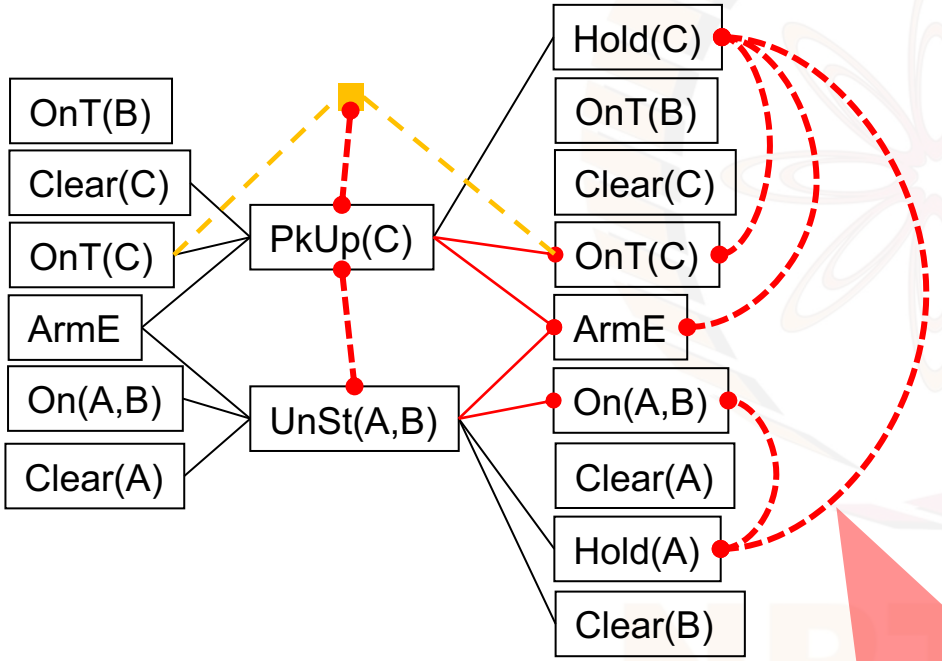
$ontable(C) \in effects^{-}(pickup(C))$

and

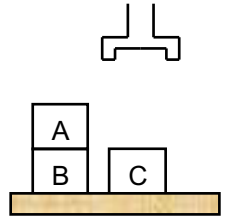
$ontable(C) \in effects^{+}(no\text{-}op)$



Mutex relations



Two propositions $p \in P_i$ and $q \in P_i$ are mutex if *all* combinations of actions <a,b> in A_i with $p \in effects^+(a)$ and $q \in effects^+(b)$ are mutex.



Some mutex propositions

The planning graph

The planning graph is made up of the following sets associated with each index *i*.

- The set of actions A_i in the i^{th} layer.
- The set of propositions P_i in the i^{th} layer.
- The set of positive effect links $PostP_i$ of actions in the i^{th} layer.
- The set of negative effect links $PostN_i$ of actions in the i^{th} layer.
- The set of preconditions links $PreP_i$ of actions in the i^{th} layer from P_{i-1}.
- The set of action *mutexes* in the i^{th} layer.
- The set of proposition *mutexes* in the i^{th} layer.

Growing the planning graph

Observe that if two propositions are non-mutex in a layer, they will be non-mutex in all subsequent layers because of the *No-op* actions.

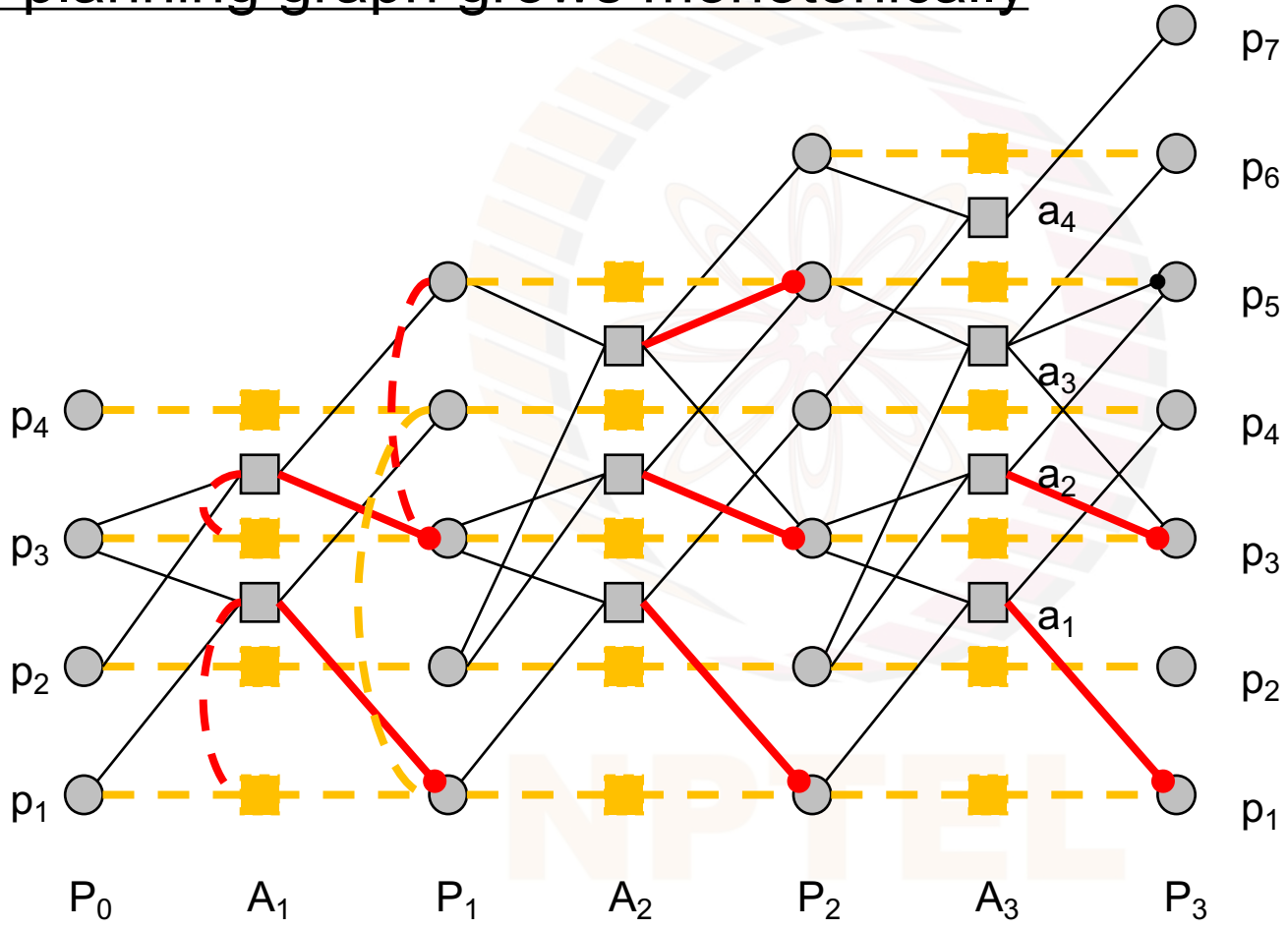
Likewise for two actions that are non-mutex.

In P_0 all propositions are non-mutex, since P_0 depicts the start state

As the graph grows with more layers being added

- the number of propositions grows monotonically
- the number of actions grows monotonically
- the number of mutexes first increases from zero
- and then decreases

The planning graph grows monotonically



Growing the planning graph

The process of extending the graph continues till any one of the following two conditions is achieved.

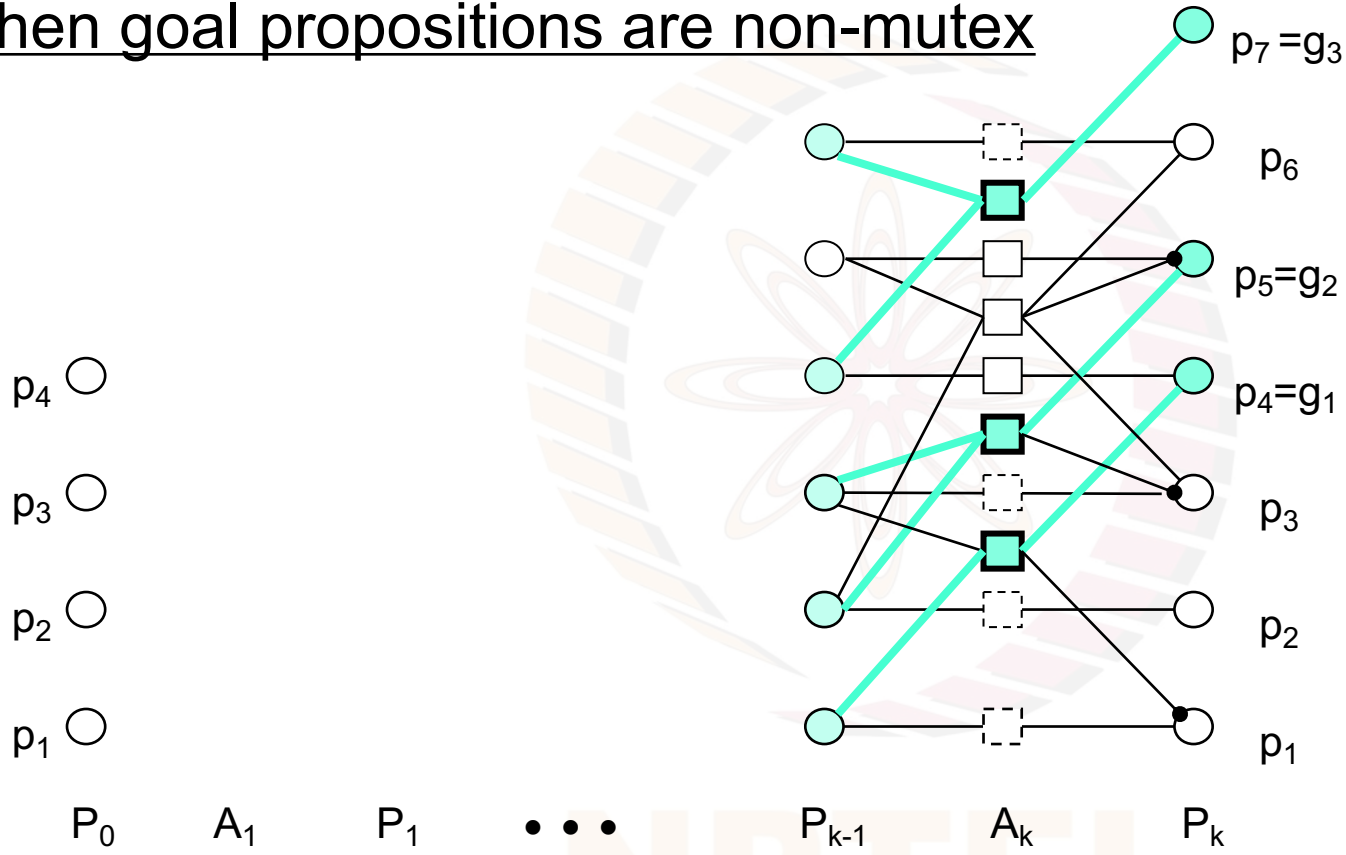
1. The newest proposition layer contains all the goal propositions, and there is no mutex relation between any of the goal propositions.

2. The planning graph has leveled off. This means that for two consecutive levels,

   ($P_{i-1} = P_i$ and $MuP_{i-1} = MuP_i$)

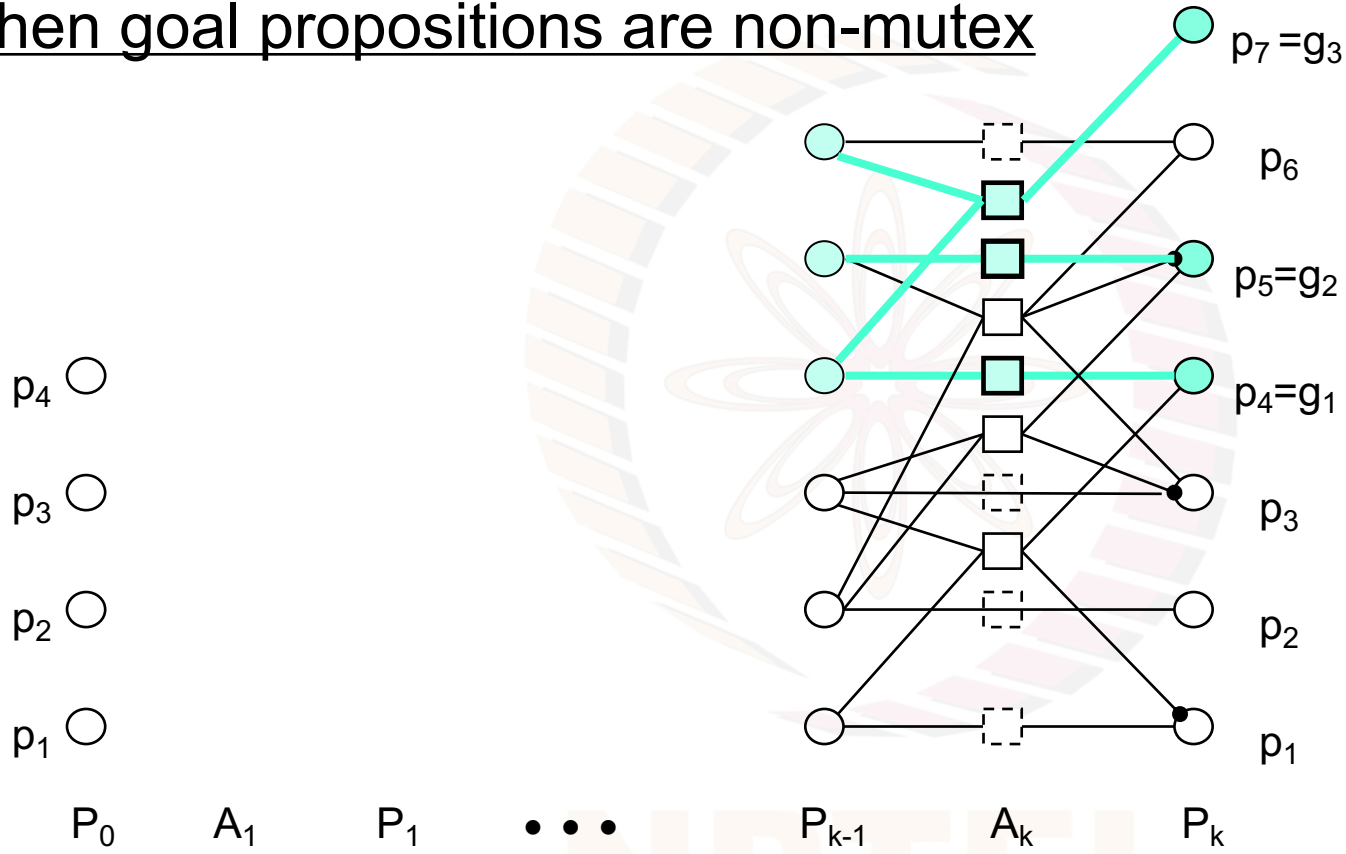
   If two consecutive levels have the same set of propositions with the same set of mutex relations between them, it means that no new actions can make an appearance. Hence if the goal propositions are not present in a levelled planning graph, they can never appear, and the problem has no solution.

When goal propositions are non-mutex



One possible subgoal set at level (k-1) is $\{p_1, p_2, p_3, p_4, p_6\}$

When goal propositions are non-mutex



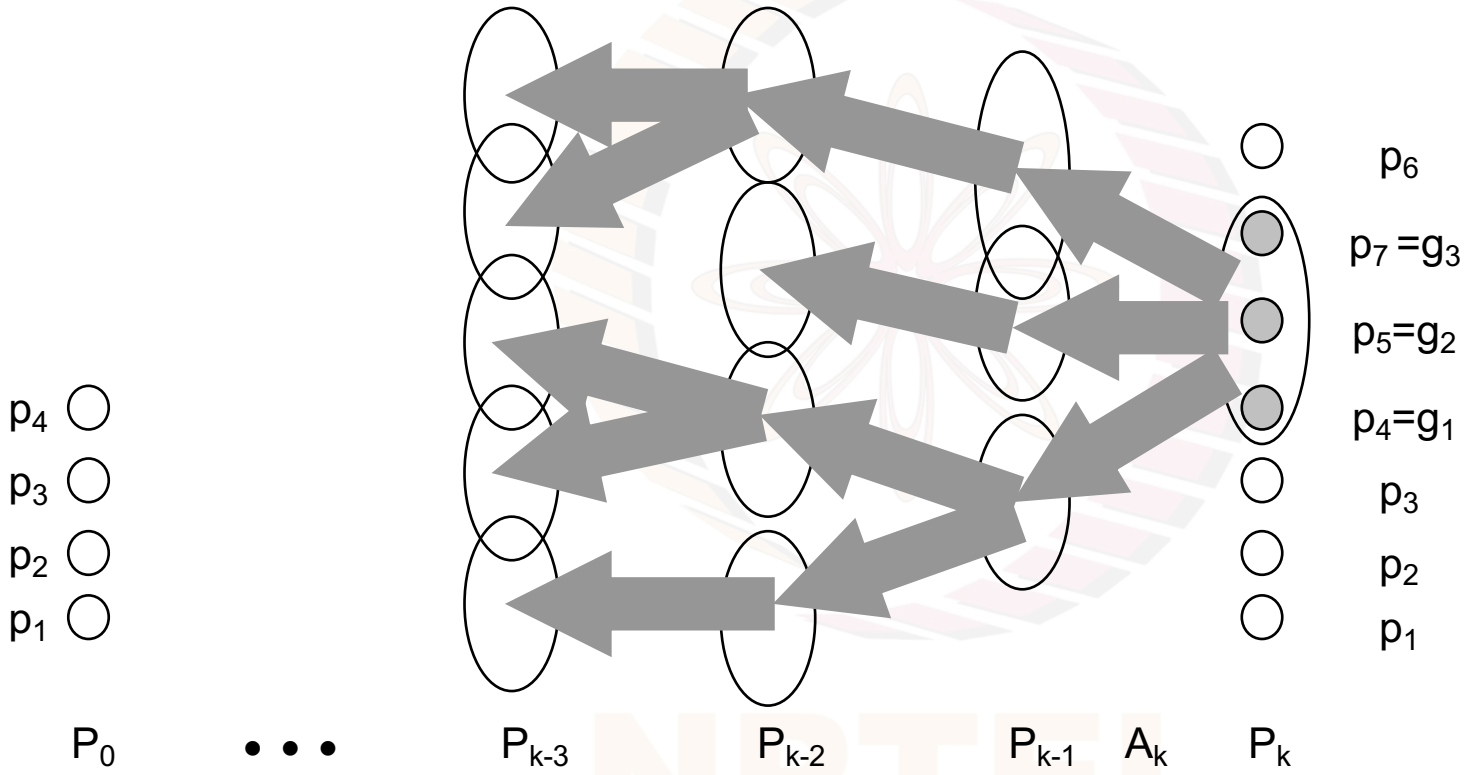
Another possible subgoal set at level (k-1) is $\{p_4, p_5, p_6\}$

When the goals are found non-mutex

When a planning graph with all goal propositions non-mutex is found

- it is possible, but not necessary, that a valid plan might exist in the graph.
- the algorithm regresses to a set of sub-goals that is non-mutex, and continues this process in a depth first fashion
- if it cannot find a sub-goal set at some level searching backwards, it backtracks and tries another sub-goal set
- if it reaches the layer P_0 at some point it returns the subgraph that is the shortest makespan plan
- else it extends the planning graph by one more level
  - this happens till the planning graph has levelled off
  - that is $P_{n-1} = P_n$ and $MuP_{n-1} = MuP_n$.
  - then it returns "*nix*".

GraphPlan does depth first search on the plannimg graph



End

Planning