Planning with State-Space Search

Planning with State-Space Search

- The most straightforward approach of planning algorithm is State Space Search
  - Forward state-space search (Progression)
  - Backward state-space search (Regression)
- Forward and backward ***state-space searches*** are forms of totally ordered plan search.

PDDL Example

- Action schema:
  $Action(Fly(p, from, to),$
  $PRECOND:$
  $EFFECT:$
- Action instantiation:
  $Action(Fly(P_1, SFO, JFK),$
  $PRECOND:$
  $EFFECT:$

PDDL Example

- Action schema:
  $Action(Fly(p, from, to),$
  $PRECOND : At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
  $EFFECT \quad : \neg At(p, from) \wedge At(p, to))$
- Action instantiation:
  $Action(Fly(P_1, SFO, JFK),$
  $PRECOND : At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK)$
  $EFFECT \quad : \neg At(P_1, SFO) \wedge At(P_1, JFK))$

State-Transition Systems

A state-transition system is a 3-tuple $\Sigma = (S,A,\gamma)$, where:

- S = set of states;
- A = set of actions;
- γ = a state transition function.

Planning problem

Given a planning problem $P=(\Sigma, s_i, S_g)$ where

- $\Sigma = (S, A, \gamma)$ is a state transition system,
- $s_i \in S$ is the initial state, and
- $S_g \subset S$ is a set of goal states,
- Find ?

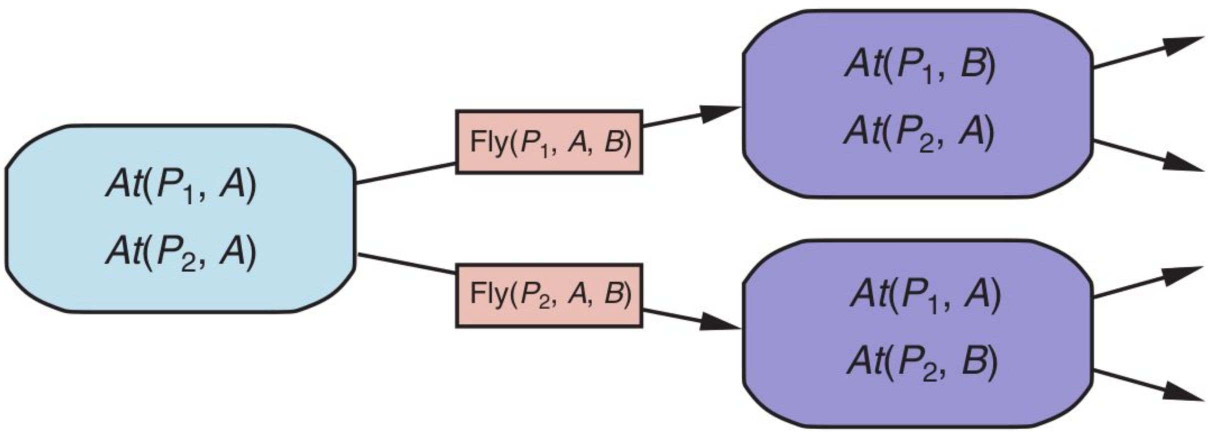
Planning problem

Given a planning problem $P=(\Sigma, s_i, S_g)$ where

- $\Sigma = (S, A, \gamma)$ is a state transition system,
- $s_i \in S$ is the initial state, and
- $S_g \subset S$ is a set of goal states,
- Find a sequence of actions $\langle a1, a2, \ldots, ak \rangle$ corresponding to a sequence of state transitions $\langle s_i, s1, \ldots, sk \rangle$ such that
- $s1 = \gamma(s_i, a1)$, $s2 = \gamma(s1, a2), \ldots$, $sk = \gamma(sk\text{-}1, ak)$, and $sk \in S_g$.

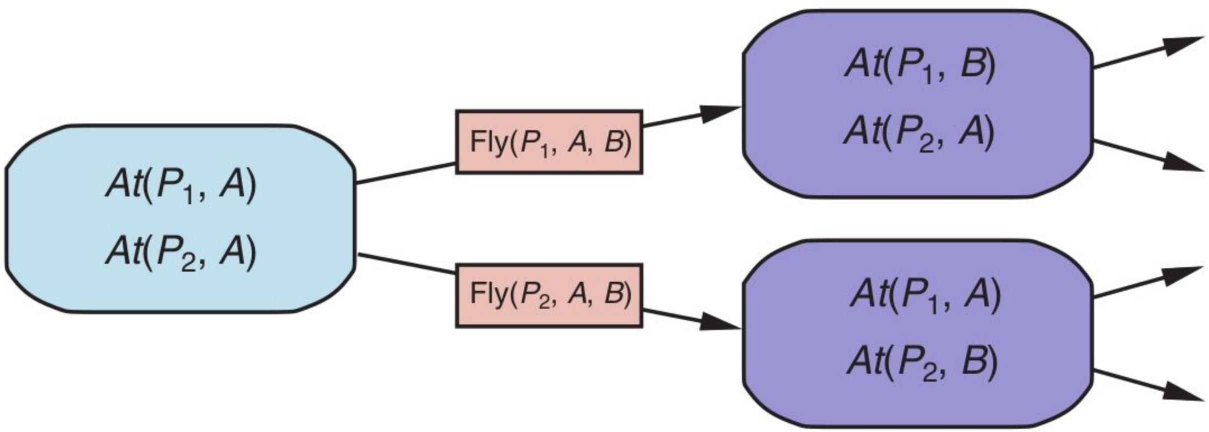
Forward Search

- **Forward Search**: starting in the initial state and using the problem's actions to search forward for a member of the set of goal states.



Forward Search

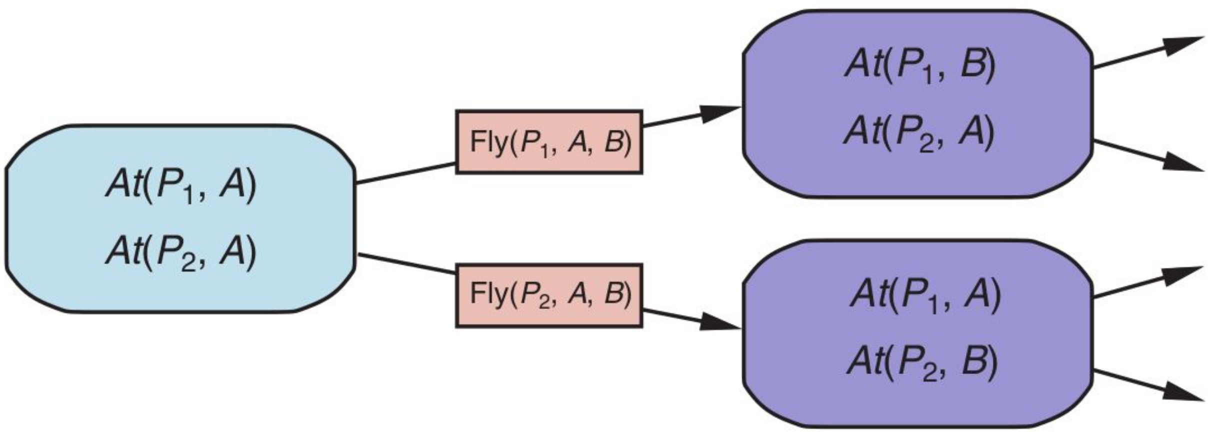
- **Forward Search**: starting in the initial state and using the problem's actions to search forward for a member of the set of goal states.



- Forward-search is sound as for any plan returned is guaranteed to be a solution

Forward Search

- **Forward Search**: starting in the initial state and using the problem's actions to search forward for a member of the set of goal states.



- Forward-search is sound as for any plan returned is guaranteed to be a solution
- Forward-search also is complete because if a solution exists then at least one of Forward search's will return a solution

Forward State-Space Search Algorithm

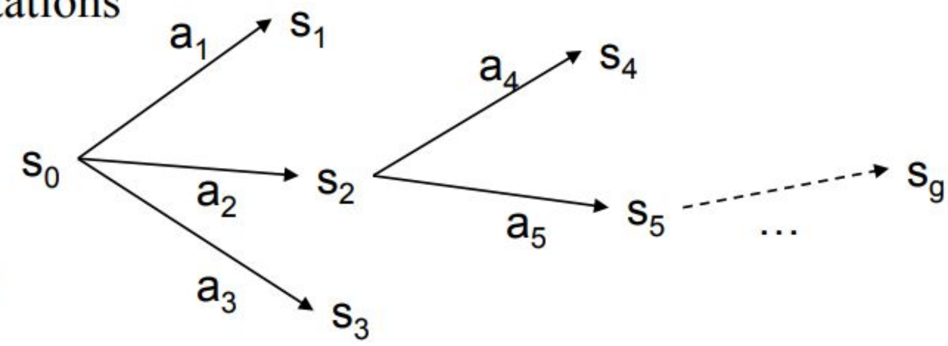
Forward-search(O, s0, g) *(O contains a list of actions)*

```
s = s0
P = the empty plan
loop
    if s satisfies g then return P
    applicable = {a | a is an operator in O, and precond(a) is true in s}
    if applicable = ∅ then return failure
    nondeterministically choose an action a from applicable
    s = γ(s, a)                    state-transition function
    P = P.a
```

Forward State-Space Search Algorithm

Some deterministic implementations of forward search:

- breadth-first search
- depth-first search
- best-first search (e.g., A*)
- greedy search



Forward State-Space Search Algorithm

Breadth-first and best-first search are sound and complete

- But they usually aren't practical because they require too much memory
- Memory requirement is exponential in the length of the solution

Forward State-Space Search Algorithm

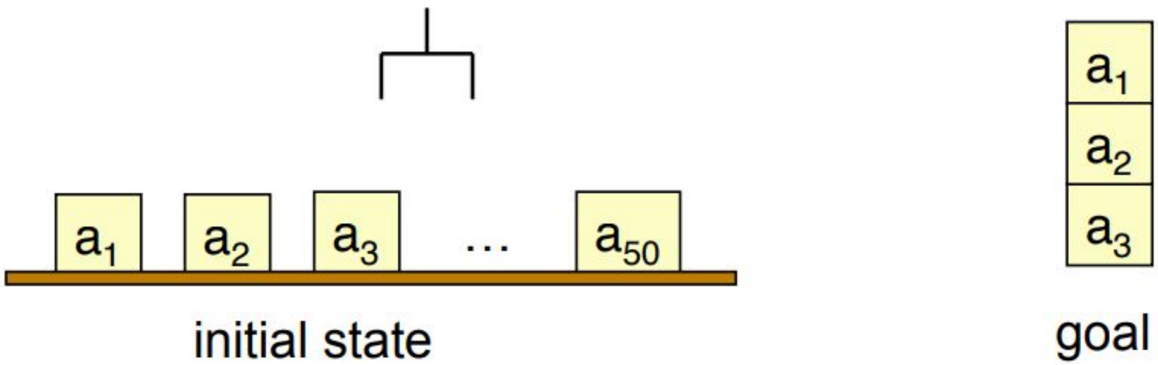
Breadth-first and best-first search are sound and complete

- But they usually aren't practical because they require too much memory
- Memory requirement is exponential in the length of the solution

In practice, more likely to use depth-first search or greedy search

- Worst-case memory requirement is linear in the length of the solution
- In general, sound but not complete
  - But classical planning has only finitely many states
  - Thus, can make depth-first search complete by doing loop-checking

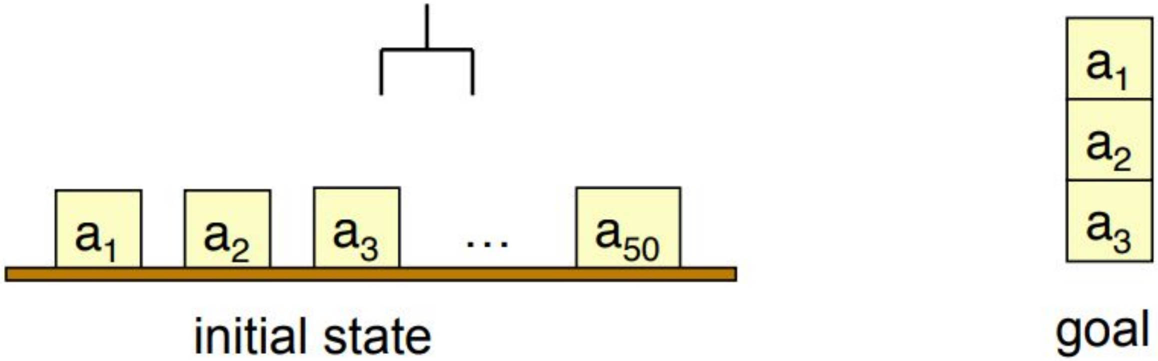
Branching Factor of Forward Search



Forward search can have a very large branching factor

- E.g., many applicable actions that don't progress toward goal

Branching Factor of Forward Search



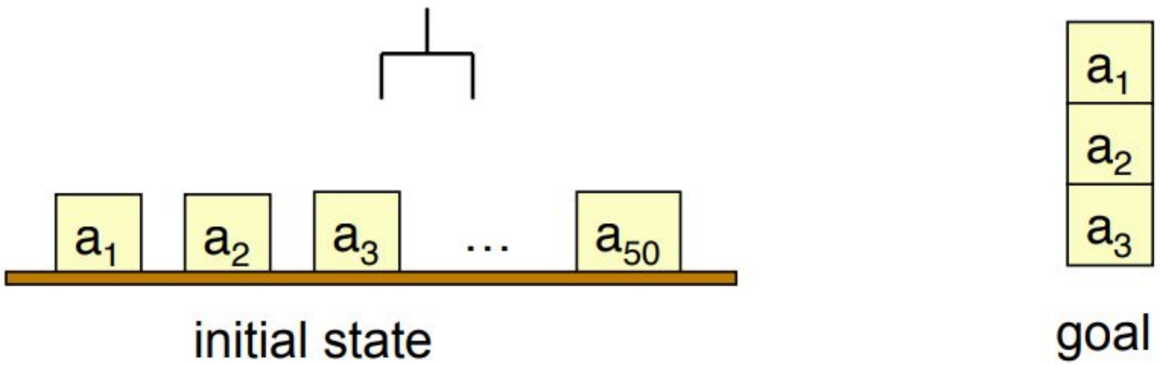
Forward search can have a very large branching factor

- E.g., many applicable actions that don't progress toward goal

Why this is bad:

- Deterministic implementations can waste time trying lots of irrelevant actions

Branching Factor of Forward Search



Forward search can have a very large branching factor

- E.g., many applicable actions that don't progress toward goal

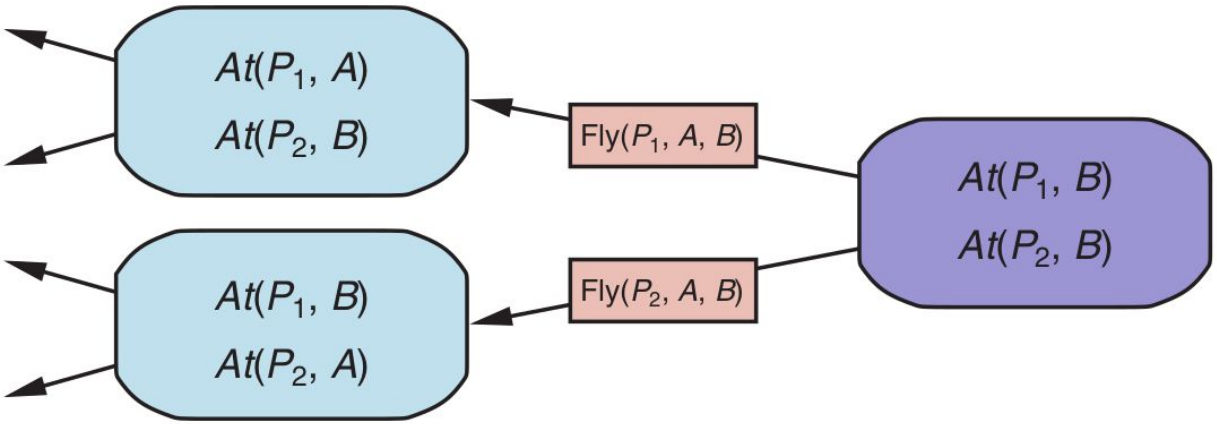
Why this is bad:

- Deterministic implementations can waste time trying lots of irrelevant actions

Need a good heuristic function and/or pruning procedure

Backward Search

- **Backward Search:** start at the goal and apply the actions backward until we find a sequence of steps that reaches the initial state.



Backward State-Space Search Algorithm

Backward-search(O, s0, g)

$s = s_0$

P = the empty plan

loop

if s0 satisfies g then return P

relevant = {a | a is an operator in O that is relevant for g}

if relevant = $\varnothing$ then return failure

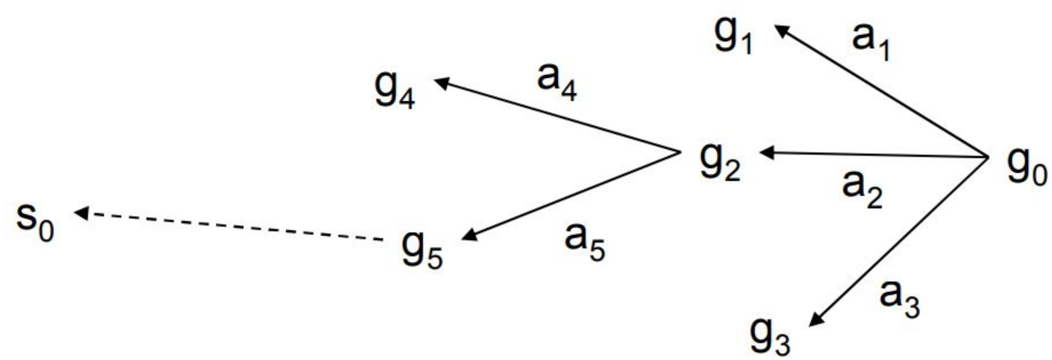
nondeterministically choose an action a from relevant

P = a.P

$g = \boldsymbol{\gamma}^{-1}(g, a)$ *inverse state transitions (new set of subgoals)*

Backward State-Space Search Algorithm

Backward-search(O, s0, g)



s = s_0

P = the empty plan

loop

if s0 satisfies g then return P

relevant = {a | a is an operator in O that is relevant for g}

if relevant = ∅ then return failure

nondeterministically choose an action a from relevant

P = a.P

g = $\boldsymbol{\gamma}^{-1}$(g, a) *inverse state transitions (new set of subgoals)*

Inverse State Transitions

If a is relevant for g, then

- $\gamma^{-1}(g,a) = (g - \text{effects}(a)) \cup \text{precond}(a)$

Otherwise $\gamma^{-1}(g,a)$ is undefined

Example: suppose that

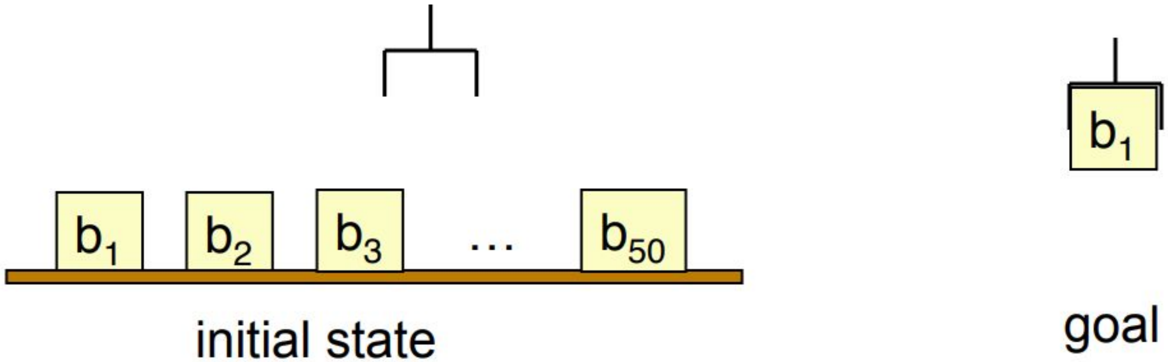
- $g = \{\text{on(b1,b2), on(b2,b3)}\}$
- $a = \text{stack(b1,b2)}$

What is $\gamma^{-1}(g,a)$?

Backward State-Space Search Algorithm

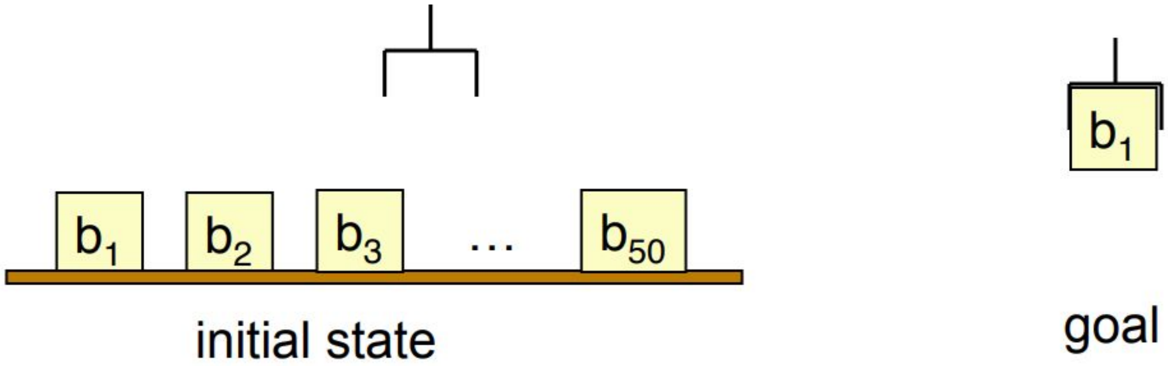
- An action a is relevant for a goal g if
  - » a makes at least one of g's literals true
    - $g \cap \text{effects}(a) \neq \emptyset$
  - » a does not make any of g's literals false
    - $g^+ \cap \text{effects}^-(a) = \emptyset$ and $g^- \cap \text{effects}^+(a) = \emptyset$

Efficiency of Backward Search



Backward search can *also* have a very large branching factor

Efficiency of Backward Search



Backward search can *also* have a very large branching factor

As before, deterministic implementations can waste lots of time trying all of them

Total-Order and Partial Order Plan

Total-Order Planning

- Total-Order Planning explore strictly linear sequences of actions, directly connected to the start or goal.
- Total-Order Planning cannot take advantages of problem decomposition.

Total-Order Planning Example

Goal(RightShoeOn ^ LeftShoeOn)
Init()

ACTION	RightShoe
PRECOND	RightSockOn
EFFECT	RightShoeOn

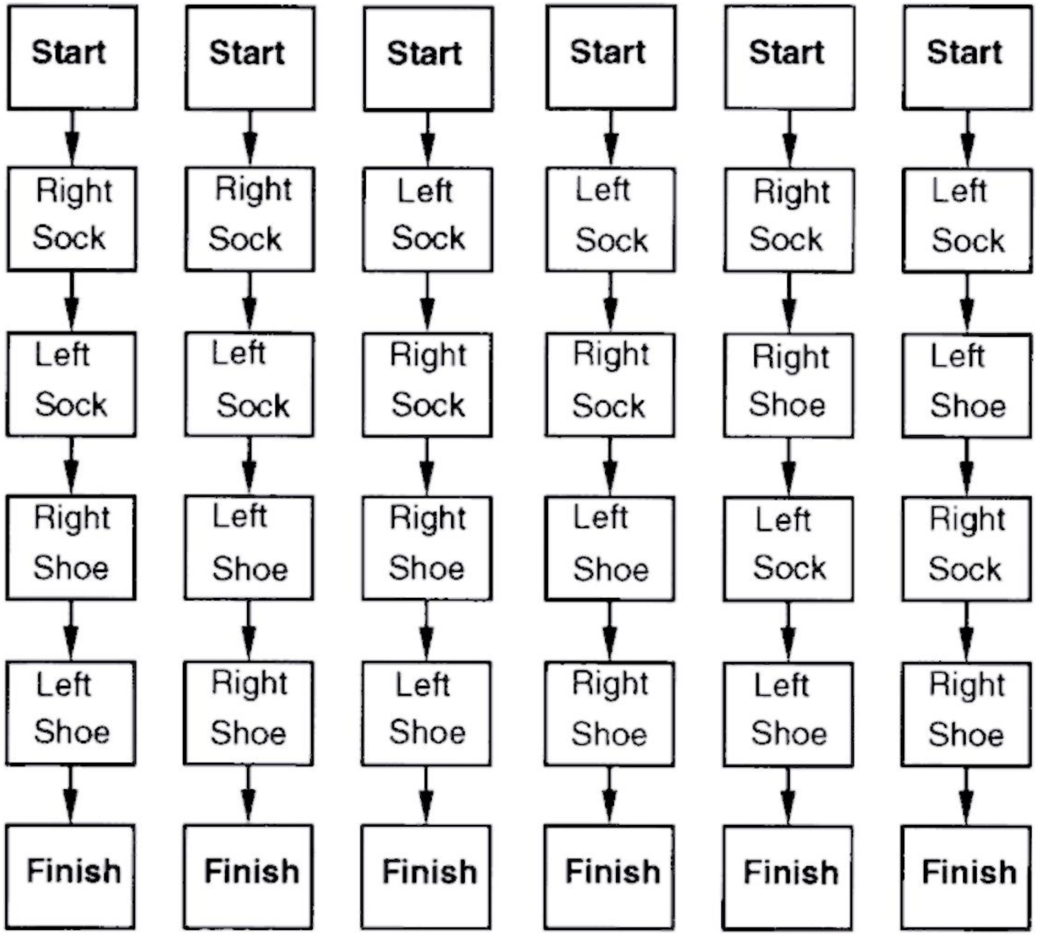
ACTION	RightSock
PRECOND	None
EFFECT	RightSockOn

ACTION	LeftShoe
PRECOND	LeftSockOn
EFFECT	LeftShoeOn

ACTION	LeftSock
PRECOND	None
EFFECT	LeftSockOn

How to define TOP for putting on a pair of shoes

Total-Order Planning Example



POP Example

Goal(RightShoeOn ^ LeftShoeOn)
Init()

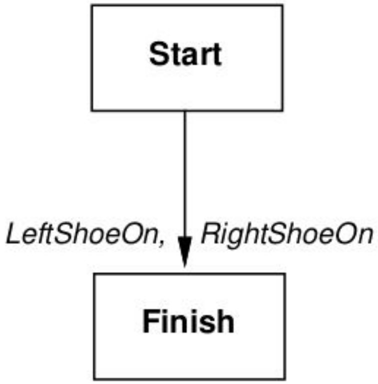
ACTION	RightShoe
PRECOND	RightSockOn
EFFECT	RightShoeOn
ACTION	RightSock
PRECOND	None
EFFECT	RightSockOn
ACTION	LeftShoe
PRECOND	LeftSockOn
EFFECT	LeftShoeOn
ACTION	LeftSock
PRECOND	None
EFFECT	LeftSockOn

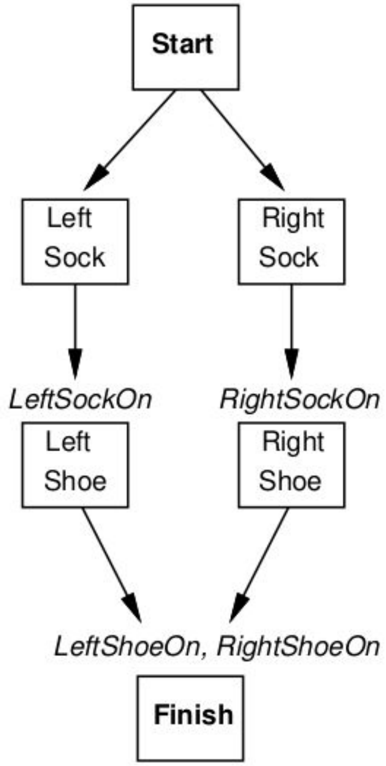
How to define POP for putting on a pair of shoes

POP Example

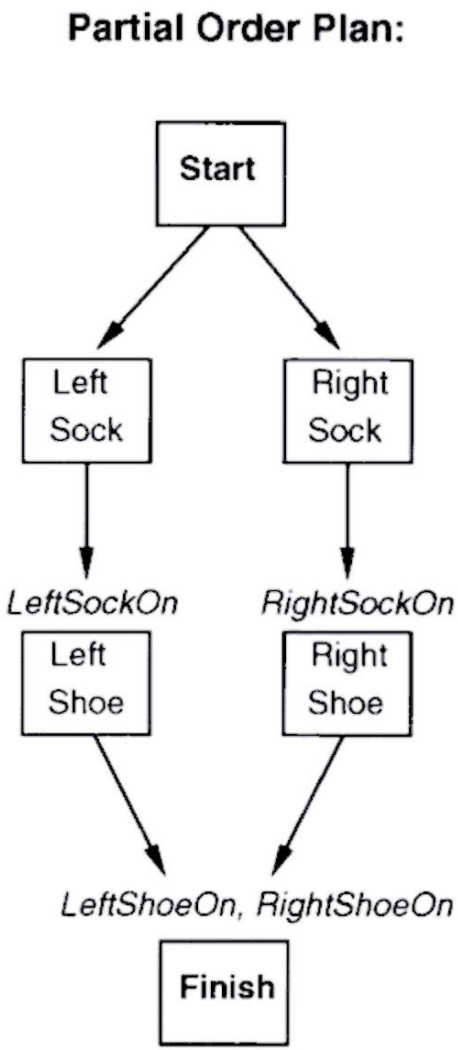
Goal(RightShoeOn ^ LeftShoeOn)
Init()

ACTION	RightShoe
PRECOND	RightSockOn
EFFECT	RightShoeOn
ACTION	RightSock
PRECOND	None
EFFECT	RightSockOn
ACTION	LeftShoe
PRECOND	LeftSockOn
EFFECT	LeftShoeOn
ACTION	LeftSock
PRECOND	None
EFFECT	LeftSockOn





POP and TOP



Partial Order Plan:
Start
Left Sock
Right Sock
LeftSockOn
RightSockOn
Left Shoe
Right Shoe
LeftShoeOn, RightShoeOn
Finish

Total Order Plans:
Start
Right Sock
Left Sock
Right Shoe
Left Shoe
Finish
Start
Right Sock
Left Sock
Left Shoe
Right Shoe
Finish
Start
Left Sock
Right Sock
Right Shoe
Left Shoe
Finish
Start
Left Sock
Right Sock
Left Shoe
Right Shoe
Finish
Start
Right Sock
Right Shoe
Left Sock
Left Shoe
Finish
Start
Left Sock
Left Shoe
Right Sock
Right Shoe
Finish

Partial-Order Planning (POP)

- POP works on several subgoals independently and solves them with sub plans. Then, combines the sub plans.
- POP specifies all actions that need to be taken, but only specifies the order between actions when necessary.
- POP has flexibility in ordering the sub plans.
- A linearization of a partial order plan is a total order plan

How to Define Partial Order Plan?

A partial-order plan consists of four components:

1. **Actions**: that make up the steps of the plan
2. **Ordering constraint**: It specifies the conditions about the order of some actions. A before B denoted as $A \prec B$
3. **Causal links**: It specifies which actions meet which preconditions of other actions. A achieves P for B is denoted as $A \xrightarrow{p} B$
4. **Open preconditions**: preconditions that are not fulfilled by any action in the partial-order plan.

The Initial Plan

- Initial plan contains:
- Start:
    - PRECOND: none
    - EFFECT: Add all propositions that are initially true
- Finish:
    - PRECOND: Goal state
    - EFFECT: none
- Ordering constraints: {}
- Causal links: {}
- Open preconditions:
    - {preconditions of Finish}

Final Plan

- The final plan has the following components:
- Actions: {RightSock, RightShoe, LeftSock, LeftShoe, Start, Finish}
- Orderings constraints: {RightSock ≺ RightShoe, LeftSock ≺ LeftShoe}
- Open preconditions: {}
- Causal Links:

$$RightSock \xrightarrow{RightSockOn} RightShoe$$

$$LeftSock \xrightarrow{LeftSockOn} LeftShoe$$

$$RightShoe \xrightarrow{RightShoeOn} Finish$$

$$LeftShoe \xrightarrow{LeftShoeOn} Finish$$

“have cake and eat cake too” problem

Init(Have(Cake) $\wedge \neg$Eaten(Cake))

Goal(Have(Cake) $\wedge$ Eaten(Cake))

Action(Eat(Cake)

PRECOND: Have(Cake)

EFFECT: $\neg$Have(Cake) $\wedge$ Eaten(Cake))

Action(Bake(Cake)

PRECOND: $\neg$Have(Cake)

EFFECT: Have(Cake)