CSD 3102 ARTIFICIAL INTELLIGENCE TECHNIQUES MODULE IV

PLANNING

Topic: Planning Problem - Simple Planning agent - Blocks world - Planning as a Statespace Search - Partial Order Planning

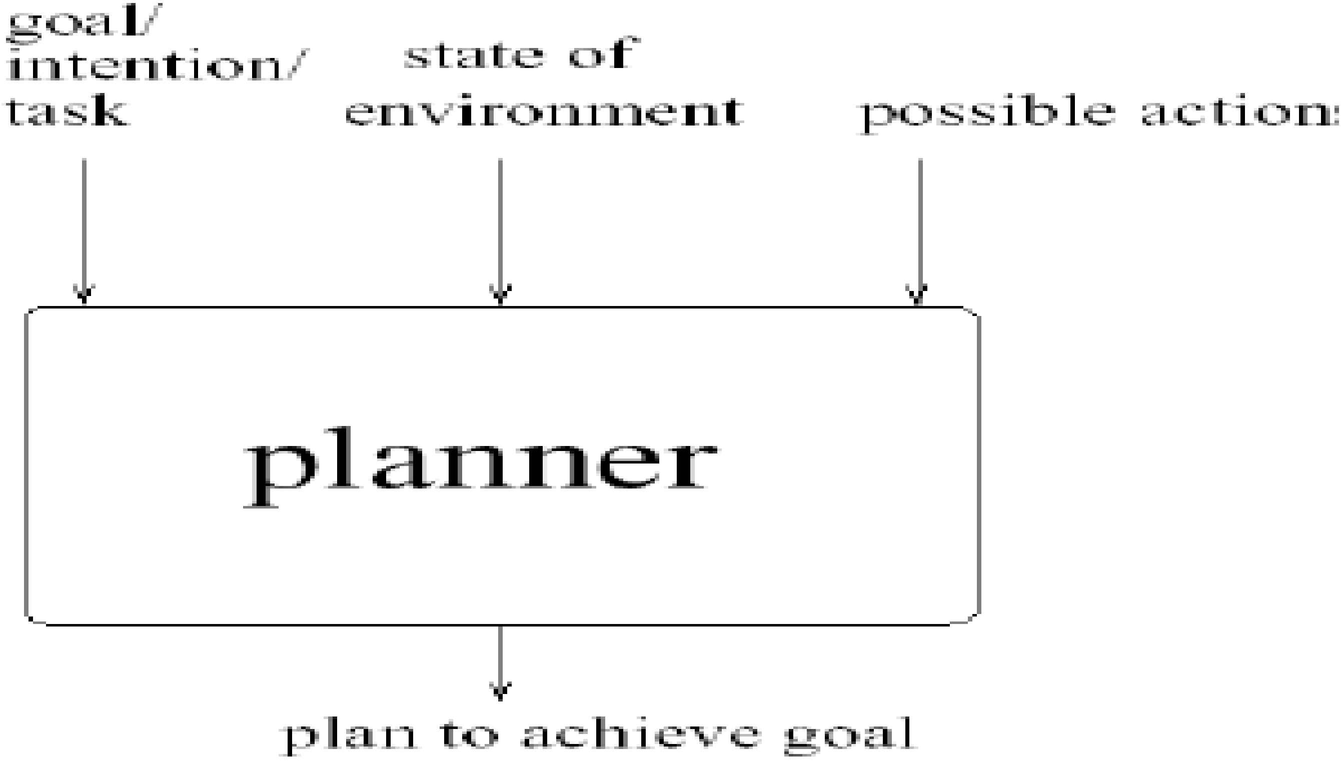
MODULE IV PLANNING 9

Planning Problem – Simple Planning agent –Blocks world - Goal Stack Planning-Means Ends Analysis- **Planning as a Statespace Search - Partial Order Planning**-Planning Graphs-Hierarchical Planning - Non- linear Planning -Conditional Planning-Reactive Planning - Knowledge based Planning-Using Temporal Logic – Execution Monitoring and Re-planning- Continuous Planning-Multi-agent Planning-Job shop Scheduling Problem.

What is a Plan?

- A sequence (list) of actions, with variables replaced by constants (specific objects in the environment)

Planner



PLANNING:

- The task of coming up with a sequence of actions that will achieve a goal is called **planning.**

Classical planning:

- We consider only environments that are fully observable, deterministic, finite, static (change happens only when the agent acts), and discrete (in time, action, objects, and effects). These are called **classical planning** environments.

Non-classical planning:

- Non-classical planning is for partially observable or stochastic environments and involves a different set of algorithms and agent designs.

PLANNING PROBLEM:

1. Overwhelmed by irrelevant actions:

- Consider the task of buying a copy of AI book: A *Modern Approach* from an online bookseller. Suppose there is one buying action for each 10-digit ISBN number, for a total of 10 billion actions.
- The search algorithm would have to examine the outcome states of all 10 billion actions to find one that satisfies the goal, which is to own a copy of ISBN 0137903952. an explicit goal description such as, **Have(ISBN0137903952)** and generate the action **Buy(ISBN0137903952).**

2. Finding a good heuristic function:

- Suppose the agent's goal is to buy four different books online. Then there will be 1040 plans of just four steps, so searching without an accurate heuristic is difficult. It requires a human to supply a heuristic function for each new problem. For the book-buying problem, the goal would be Have(A) ∧ Have (B) ∧ Have(C) ∧ Have(D) , and a state containing Have (A) ∧ Have(C) would have cost 2. Thus, the agent automatically gets the right heuristic for this problem.

3. Problem decomposition:

- The problem solver might be inefficient because it cannot take advantage of **problem decomposition.**
- Consider the problem of delivering a set of overnight packages to their respective destinations, which are scattered across Australia.
- It makes sense to find out the nearest airport for each destination and divide the overall problem into several subproblems, one for each airport.

THE LANGUAGE OF PLANNING PROBLEMS:

- The representation of planning problems-**states, actions, and goals**-should make it possible for planning algorithms to take advantage of the logical structure of the problem.
- The basic representation language of classical planners, known as the **STRIPS(Stanford Research Institute Problem Solver) language.**
- STRIPS is insufficiently expressive for some real domains where **Action Description Language (ADL)** can be used.
- Planning consists of following components:

❑**Representation of states.** Planners decompose the world into logical conditions and represent a state as a conjunction of positive literals. We will consider propositional literals;

❑**Representation of goals. A** goal is a partially specified state, represented as a conjunction of positive ground literals.

❑**Representation of actions.** An action is specified in terms of the preconditions that must hold before it can be executed and the effects that ensue when it is executed.

THE LANGUAGE OF PLANNING PROBLEMS:

In general, an action schema consists of three parts:

- The **action name and parameter** list-for example, Fly(p, from, to)-serves to identify the action.
- The **precondition** is a conjunction of function-free positive literals stating which must be true in a state before the action can be executed. Any variables in the precondition must also appear in the action's parameter list.
- The **effect** is a conjunction of function-free literals describing how the state changes when the action is executed. A positive literal P in the effect is asserted to be true in the state resulting from the action, whereas a negative literal **⌐***P* is asserted to be false. Variables in the effect must also appear in the action's parameter list.

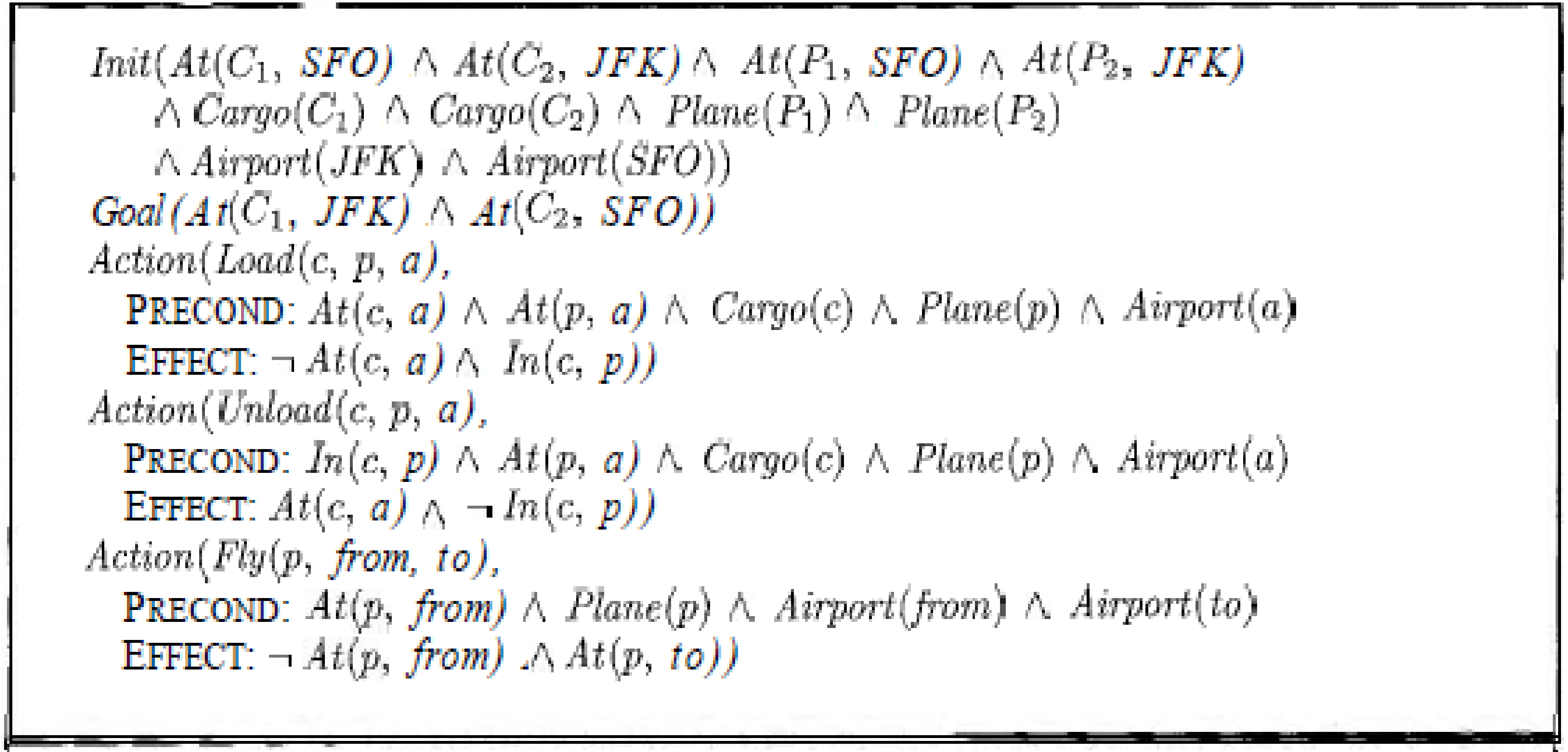
STRIPS Examples:

Example: Air cargo transport

An air cargo transport problem involving loading and unloading cargo onto and off of planes and flying it from place to place. The problem can be defined with three actions: *Load, Unload,* and *Fly.*

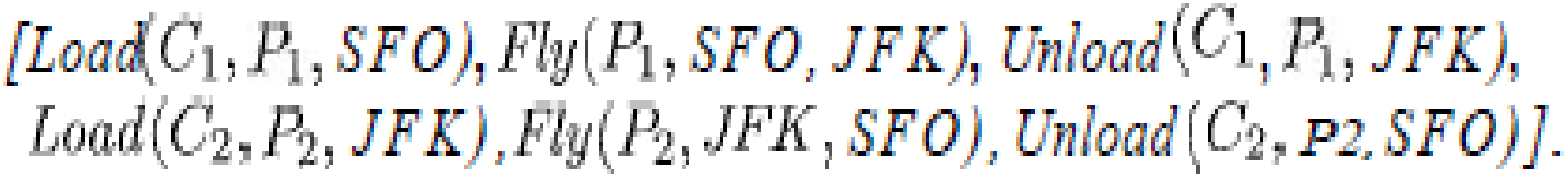
The actions affect two predicates: *In(c,* p) means that cargo c is inside plane p, and *A t (x , a)* means that object x (either plane or cargo) is at airport a.

A STRIPS problem involving transportation of air cargo between airports..



$Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK)$
$\quad \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2)$
$\quad \land Airport(JFK) \land Airport(SFO))$
$Goal(At(C_1, JFK) \land At(C_2, SFO))$
$Action(Load(c, p, a),$
$\quad$ PRECOND: $At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
$\quad$ EFFECT: $\neg At(c, a) \land In(c, p))$
$Action(Unload(c, p, a),$
$\quad$ PRECOND: $In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
$\quad$ EFFECT: $At(c, a) \land \neg In(c, p))$
$Action(Fly(p, from, to),$
$\quad$ PRECOND: $At(p, from) \land Plane(p) \land Airport(from) \land Airport(to)$
$\quad$ EFFECT: $\neg At(p, from) \land At(p, to))$

- The following plan is a solution to the Air Cargo Transport problem:



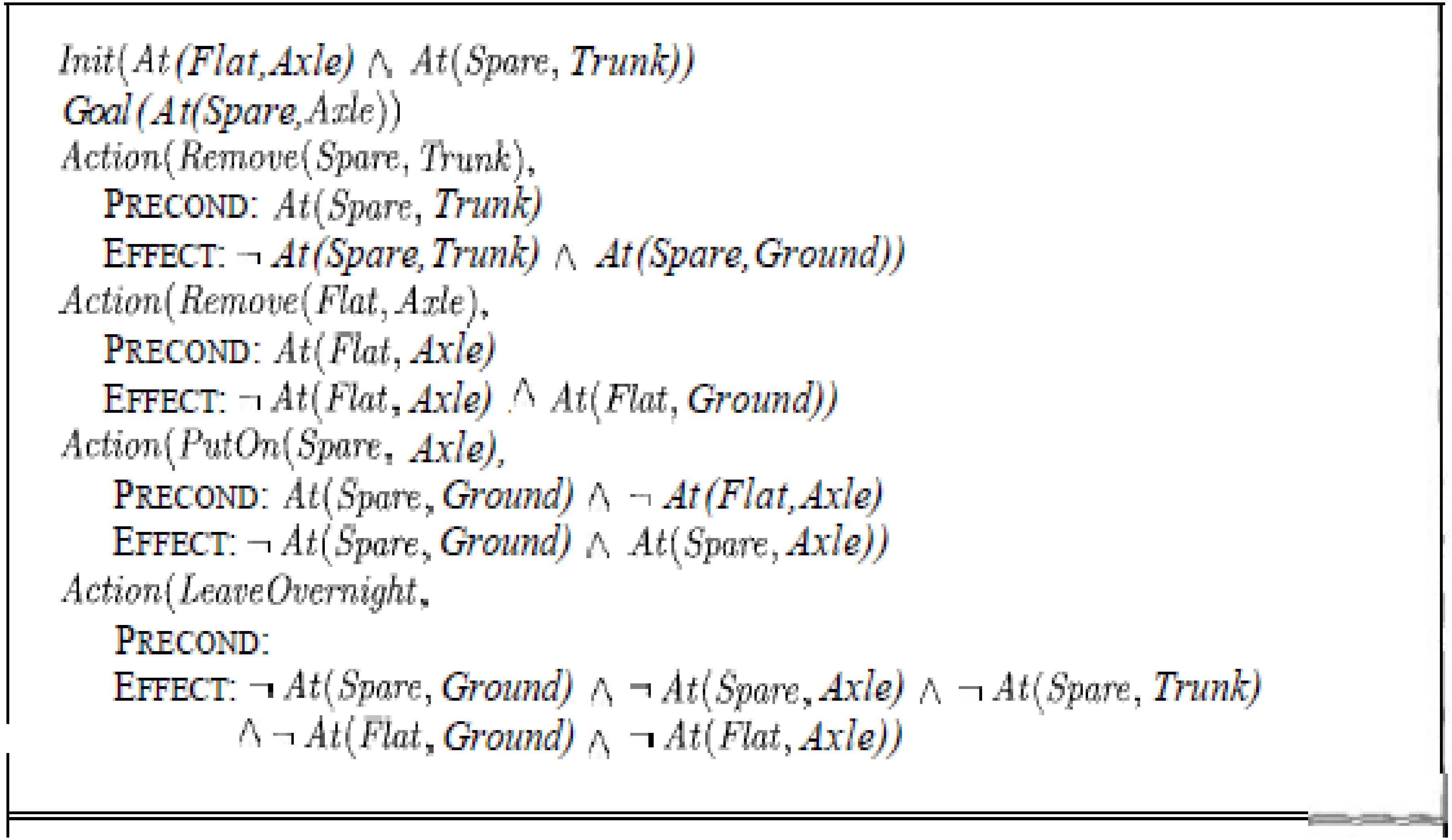
$$[Load(C_1, P_1, SFO), Fly(P_1, SFO, JFK), Unload(C_1, P_1, JFK),$$
$$Load(C_2, P_2, JFK), Fly(P_2, JFK, SFO), Unload(C_2, P2, SFO)].$$

Example: The spare tire problem

Consider the problem of changing a flat tire. More precisely, the goal is to have a good spare tire properly mounted onto the car's axle, where the initial state has a flat tire on the axle and a good spare tire in the trunk. There are just four actions:

- Removing the spare from the trunk
- Removing the flat tire from the axle
- Putting the spare on the axle
- Leaving the car unattended overnight.
- We assume that the car is in a particularly bad neighborhood, so that the: effect of leaving it overnight is that the tires disappear.

Example: The spare tire problem



$Init(At(Flat, Axle) \land At(Spare, Trunk))$

$Goal(At(Spare, Axle))$

$Action(Remove(Spare, Trunk),$

$\quad \textsc{Precond}: At(Spare, Trunk)$

$\quad \textsc{Effect}: \neg At(Spare, Trunk) \land At(Spare, Ground))$

$Action(Remove(Flat, Axle),$

$\quad \textsc{Precond}: At(Flat, Axle)$

$\quad \textsc{Effect}: \neg At(Flat, Axle) \land At(Flat, Ground))$

$Action(PutOn(Spare, Axle),$

$\quad \textsc{Precond}: At(Spare, Ground) \land \neg At(Flat, Axle)$

$\quad \textsc{Effect}: \neg At(Spare, Ground) \land At(Spare, Axle))$

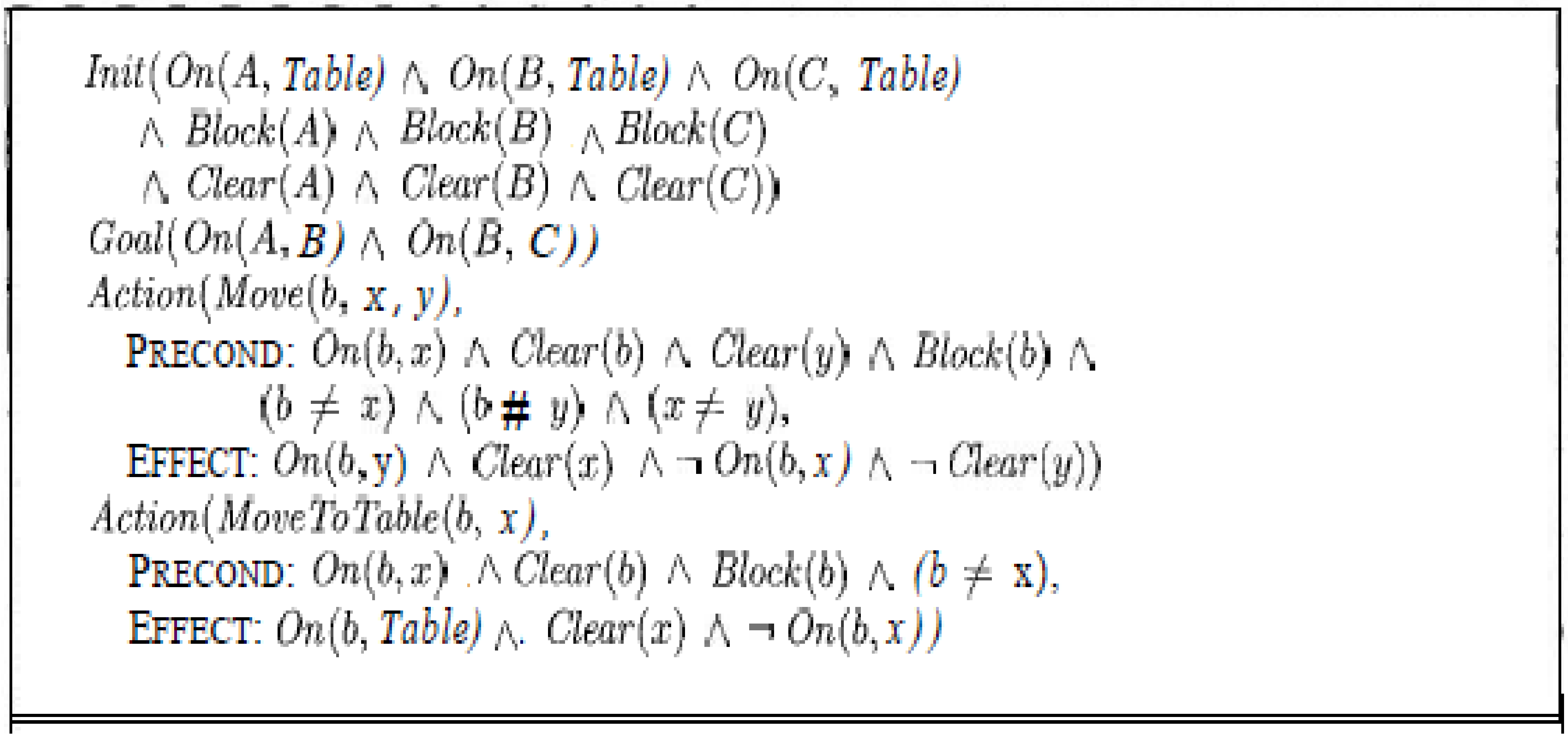
$Action(LeaveOvernight,$

$\quad \textsc{Precond}:$

$\quad \textsc{Effect}: \neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk)$

$\qquad \land \neg At(Flat, Ground) \land \neg At(Flat, Axle))$

Building Block world problem



$Init(On(A, Table) \land On(B, Table) \land On(C, Table)$

$\quad \land Block(A) \land Block(B) \land Block(C)$

$\quad \land Clear(A) \land Clear(B) \land Clear(C))$

$Goal(On(A, B) \land On(B, C))$

$Action(Move(b, x, y),$

$\quad$ PRECOND: $On(b, x) \land Clear(b) \land Clear(y) \land Block(b) \land$

$\qquad\qquad (b \neq x) \land (b \# y) \land (x \neq y),$

$\quad$ EFFECT: $On(b, y) \land Clear(x) \land \neg On(b, x) \land \neg Clear(y))$

$Action(MoveToTable(b, x),$

$\quad$ PRECOND: $On(b, x) \land Clear(b) \land Block(b) \land (b \neq x),$

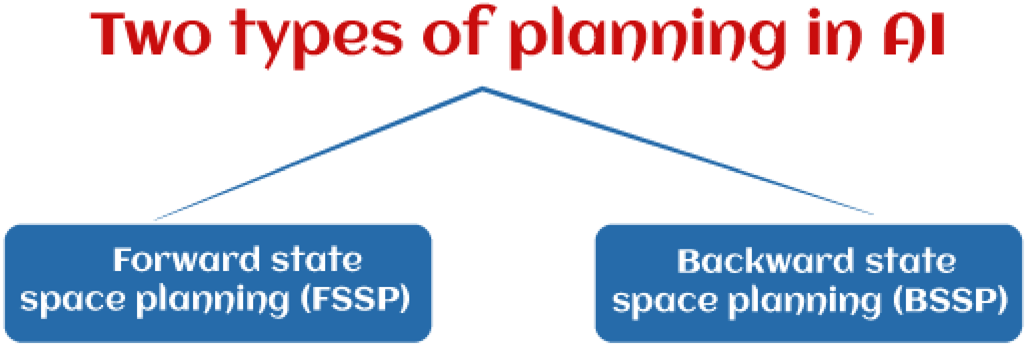
$\quad$ EFFECT: $On(b, Table) \land Clear(x) \land \neg On(b, x))$

COMPONENTS OF A PLANNING SYSTEM:

- Choose the best rule to apply the next rule based on the best available guess.
- Apply the chosen rule to calculate the new problem condition.
- Find out when a solution has been found.
- Detect dead ends so they can be discarded and direct system effort in more useful directions.
- Find out when a near-perfect solution is found.

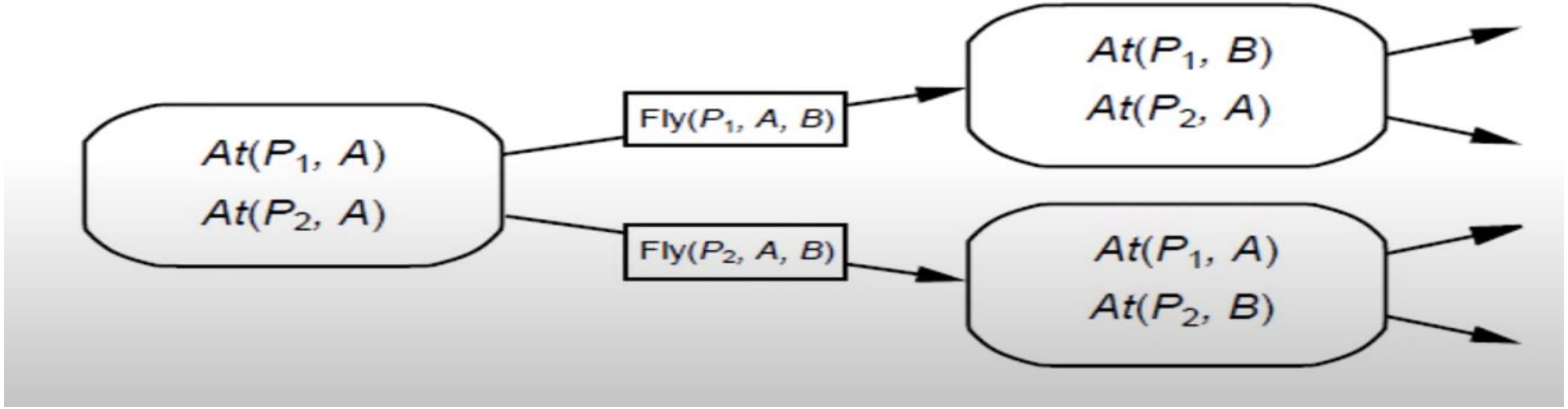
PLANNING WITH STATE SPACE SEARCH: Planning Algorithm

- The most straightforward approach is to use state-space search.
- Because the descriptions of actions in a planning problem specify both preconditions and effects, it is possible to search in either direction: either forward from the initial state or backward from the goal.
- we have **Forward State Space Planning (FSSP)** and **Backward State Space Planning (BSSP)** at the basic level.


Two types of planning in AI
Forward state space planning (FSSP)
Backward state space planning (BSSP)

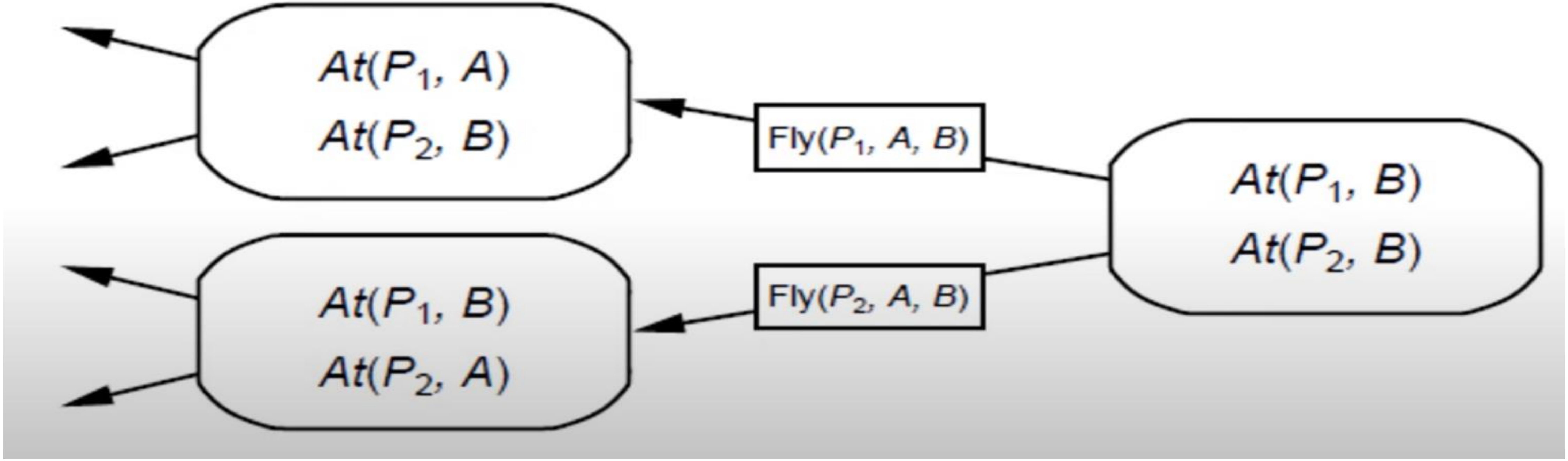
Forward State Space Planning (FSSP)

- FSSP behaves in the same way as forwarding state-space search.
- It says that given an initial state S in any domain, we perform some necessary actions and obtain a new state S' (which also contains some new terms), called a progression.
- It continues until we reach the target position. Action should be taken in this matter.



Backward State Space Planning (BSSP)

- BSSP behaves similarly to backward state-space search. In this, we move from the target state g to the sub-goal g, tracing the previous action to achieve that goal.
- This process is called regression (going back to the previous goal or sub-goal). These sub-goals should also be checked for consistency. The action should be relevant in this case.



Partial Order Planning

The Planning Problem

Formally a planning algorithm has three inputs:

1. A description of the world in some formal language,
2. A description of the agent’s goal in some formal language, and
3. A description of the possible actions that can be performed.

   The planner’s o/p is a sequence of actions which when executed in any world satisfying the initial state description will achieve the goal.

Representation for states and goals

In the STRIPS language, states are represented by conjunctions of function-free ground literals, that is, predicates applied to constant symbols, possibly negated. For example,

At(Home)^ ¬ Have(Milk)^ ¬ Have(Bananas)^ ¬ Have(Drill)^....

Goals are also described by conjunctions of literals. For example,

At(Home)^Have(Milk)^ Have(Bananas)^ Have(Drill)

Goals can also contain variables. For example, the goal of being at a store that sells milk would be represented as

Representations for actions

Our STRIPS operators consist of three components:

• the ***action description*** is what an agent actually returns to the environment in order to do something.

• the ***precondition*** is a conjunction of atoms (positive literals) that says what must be true before the operator can be applied.

• the ***effect*** of an operator is a conjunction of literals (positive or negative) that describes how the situation changes when the operator is applied.

Here’s an example for the operator for going from one place to another:

Op(Action:Go(there), Precond:At(here)^Path(here, there), Effect:At(there)^ ¬At(here))

Search through World Space

The simplest way to build a planner is to cast the planning problem as search through the space of world states. Each node in the graph denotes a state of the world, and arcs connect worlds that can be reached by executing a single action. We would call it a **situation space** planner because it searches through the space of possible situations.

There are two types of planners:

Progression Planner: it searches forward from the initial situation to the goal situation.

Regression Planner: it searches backward from the goal situation to the initial situation.

Search through the Space of Plans

An alternative is to search through space of plans rather than space of situations i.e plan-space node denote plans

- Start with a simple partial plan
- Expand the plan until the complete plan is developed
- Operators in this step:
    - Adding a step
    - Imposing an ordering that puts one step after another
    - Instantiating a previously unbound variable
- The solution → final plan   Path → irrelevant

Representation of Plans

Consider a simple problem:

- Putting on a pair of shoes

- Goal →RightShoeOn ^ LeftShoeOn
- Four operators:
  - **Op(Action:RightShoe,PreCond:RightSockOn,Effect:RightShoeON)**
  - **Op(Action:RightSock , Effect: RightSockOn)**
  - **Op(Action:LeftShoe, Precond:LeftSockOn, Effect:LeftShoeOn)**
  - **Op(Action:LeftSock,Effect:LeftSockOn)**

Least Commitment –

One should make choices only about things that you currently care about ,leaving the others to be worked out later.

Partial Order Planner –

A planner that can represent plans in which some steps are ordered (before or after) w.r.t each other and other steps are unordered.

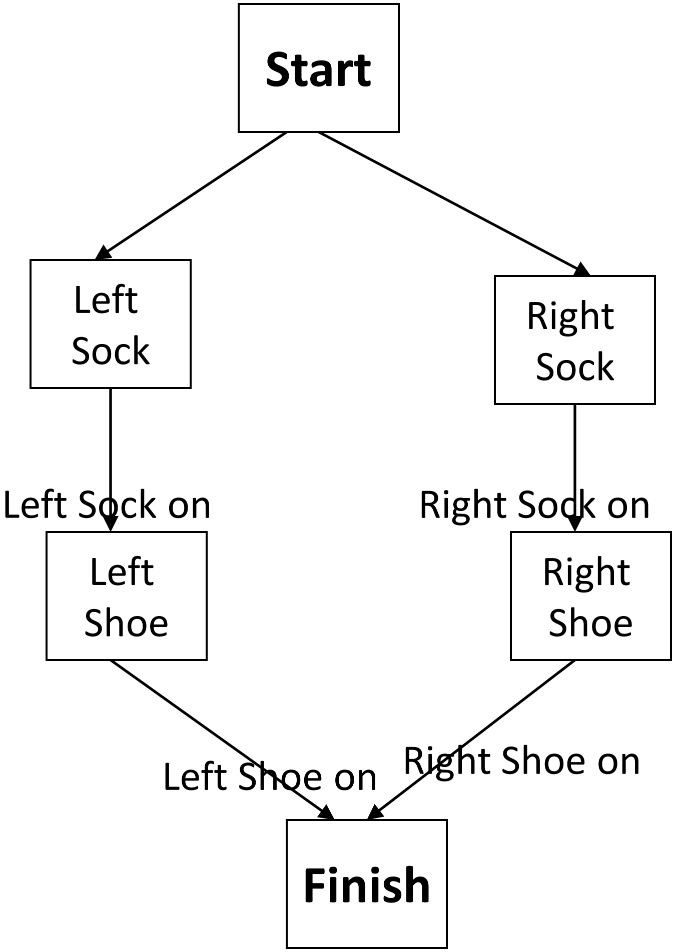
Total Order Planner––

Planner in which plans consist of a simple lists of steps

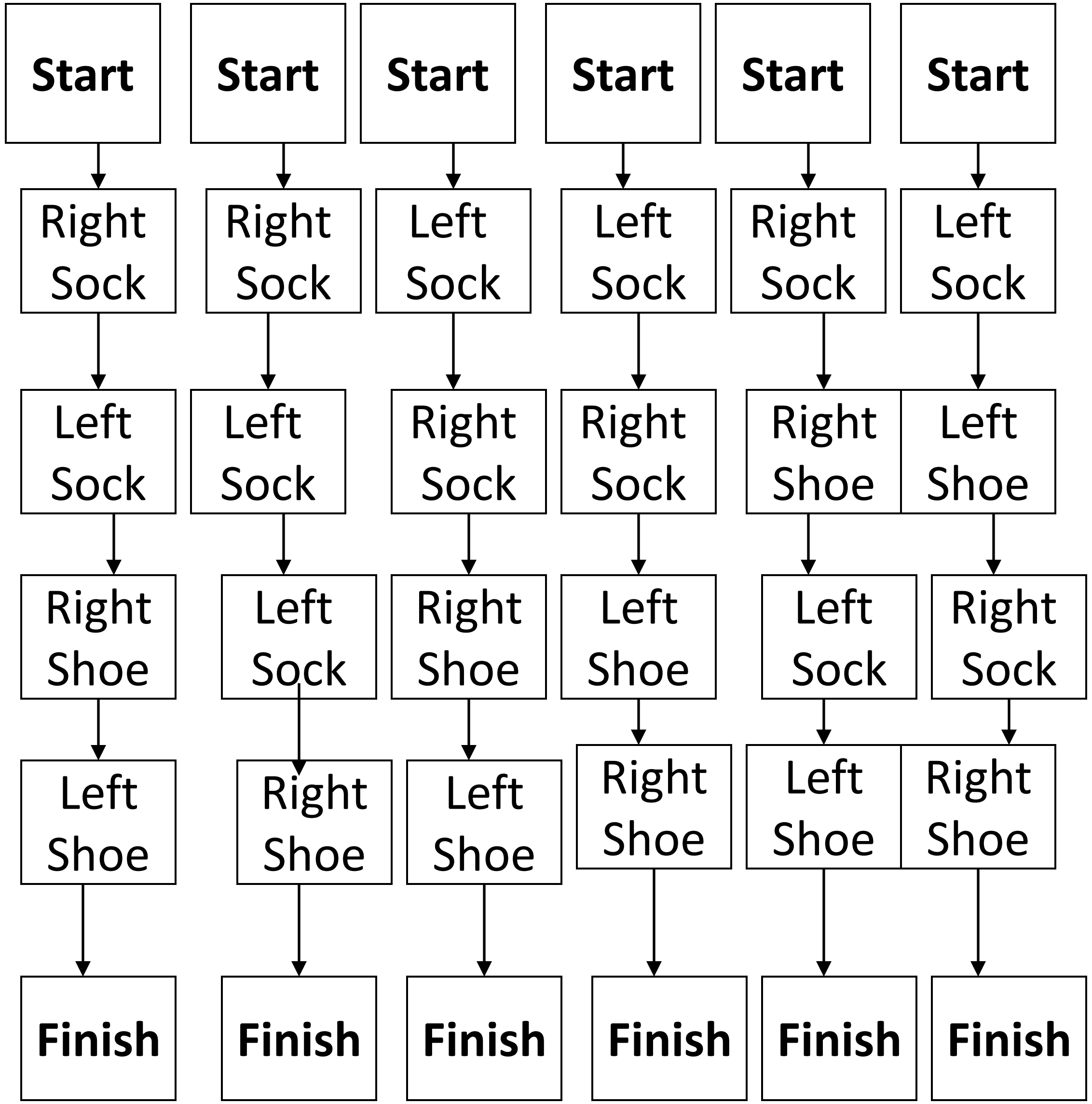
Linearization of P––

A totally ordered plan that is derived from a plan P by adding constraints

Partial Order Plans:



Total Order Plans:



PARTIAL ORDER PLANNING ALGORITHM(POP ALGORITHM)

We will define the POP algorithm for partial-order planning. Each plan has the following four components, where the first two define the steps of the plan and the last two serve a bookkeeping function to determine how plans can be extended:

- A set of **actions** that make up the steps of the plan. These are taken from the set of actions in the planning problem. The "empty" plan contains just the Start and Finish actions. Start has no preconditions and has as its effect all the literals in the initial state of the planning problem. Finish has no effects and has as its preconditions the goal literals of the planning problem.

- A set of **ordering constraints.** Each ordering constraint is of the form $A \prec B$ which is read as "A before B" and means that action A must be executed sometime before action B, but not necessarily immediately before. The ordering constraints must describe a proper partial order. Any cycle such as $A \prec B$ and $B \prec A$

The above cycle represents a contradiction, so an ordering constraint cannot be added to the plan if it creates a cycle.

- **Causal Links—**

A Causal Links is written as $S_i \xrightarrow{c} S_j$ and read as “ S_i achieves c for S_j “.Causal Links serve to record the purpose of steps in the plan:here a purpose of S_i is to achieve the precondition c of S_j

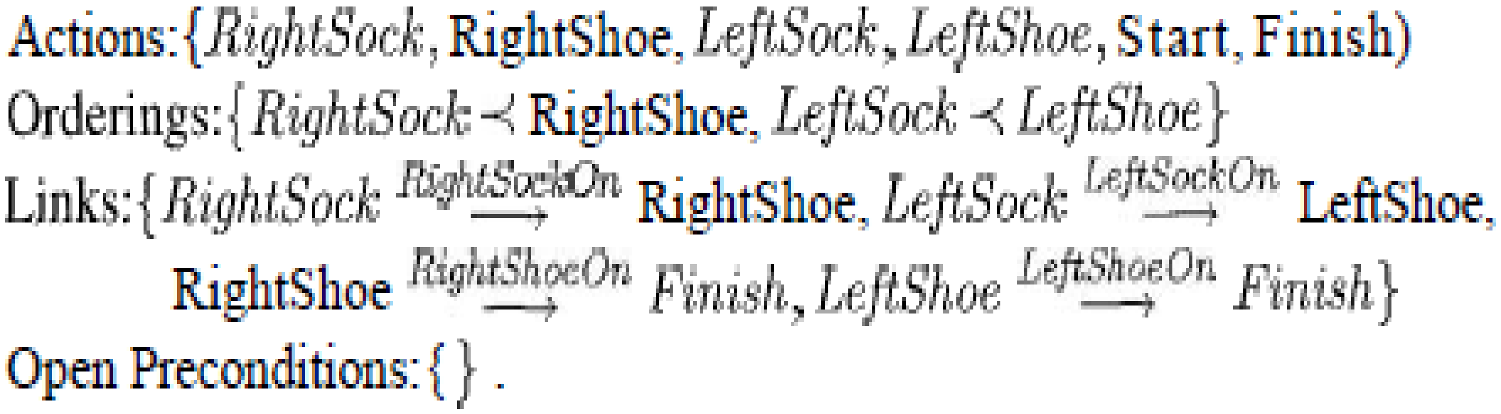
A set of **causal links.** A causal link between two actions A and B in the plan is written as A $\xrightarrow{p}$ B and is read as "A **achieves** *p* for **B."** For example, the causal link asserts that *RightSockOn* is an effect of the *RightSock* action and a precondition of *RightShoe.*

•A causal link between two actions A and B in the plan is written as A

$$RightSock \xrightarrow{RightSockOn} RightShoe$$

- A set of open preconditions. A precondition is open if it is not achieved by some action in the plan. Planners will work to reduce the set of open preconditions to the empty set, without introducing a contradiction.

For example, the final plan has the following components:

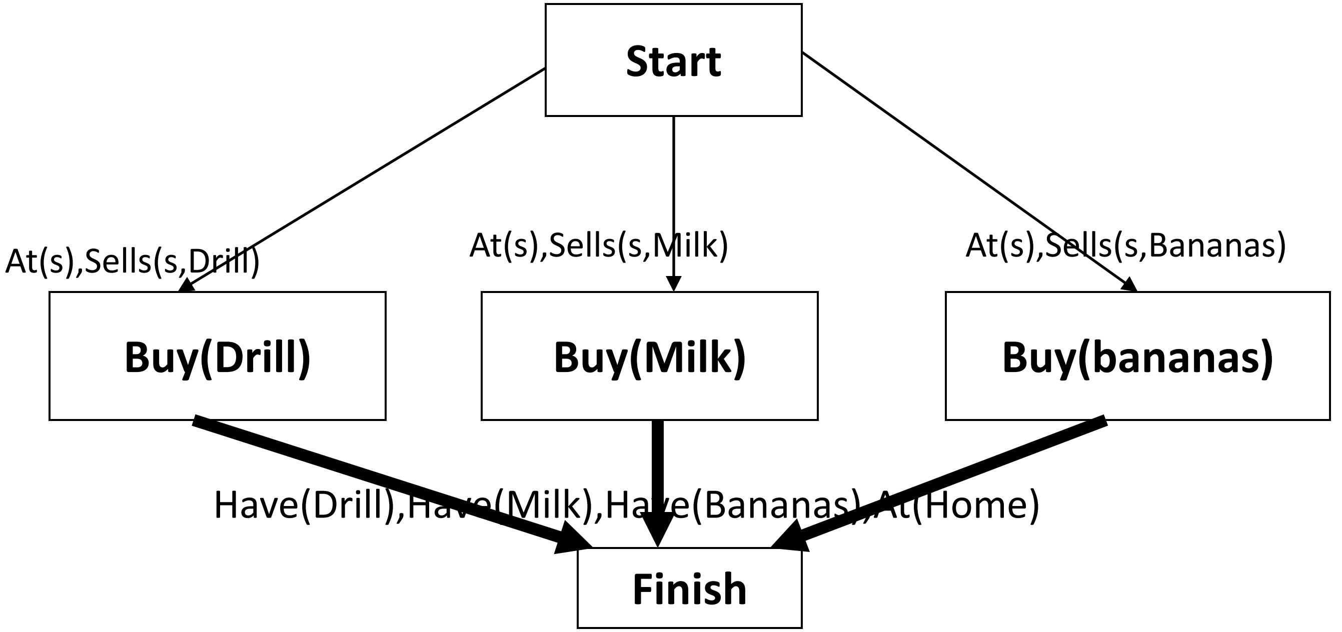


Actions:$\{\mathit{RightSock}$, RightShoe, $\mathit{LeftSock}$, $\mathit{LeftShoe}$, Start, Finish)

Orderings:$\{\mathit{RightSock} \prec$ RightShoe, $\mathit{LeftSock} \prec \mathit{LeftShoe}\}$

Links:$\{\mathit{RightSock} \xrightarrow{\mathit{RightSockOn}}$ RightShoe, $\mathit{LeftSock} \xrightarrow{\mathit{LeftSockOn}}$ LeftShoe,
RightShoe $\xrightarrow{\mathit{RightShoeOn}} \mathit{Finish}$, $\mathit{LeftShoe} \xrightarrow{\mathit{LeftShoeOn}} \mathit{Finish}\}$

Open Preconditions:$\{\ \}$.



Start

At(s),Sells(s,Drill)

At(s),Sells(s,Milk)

At(s),Sells(s,Bananas)

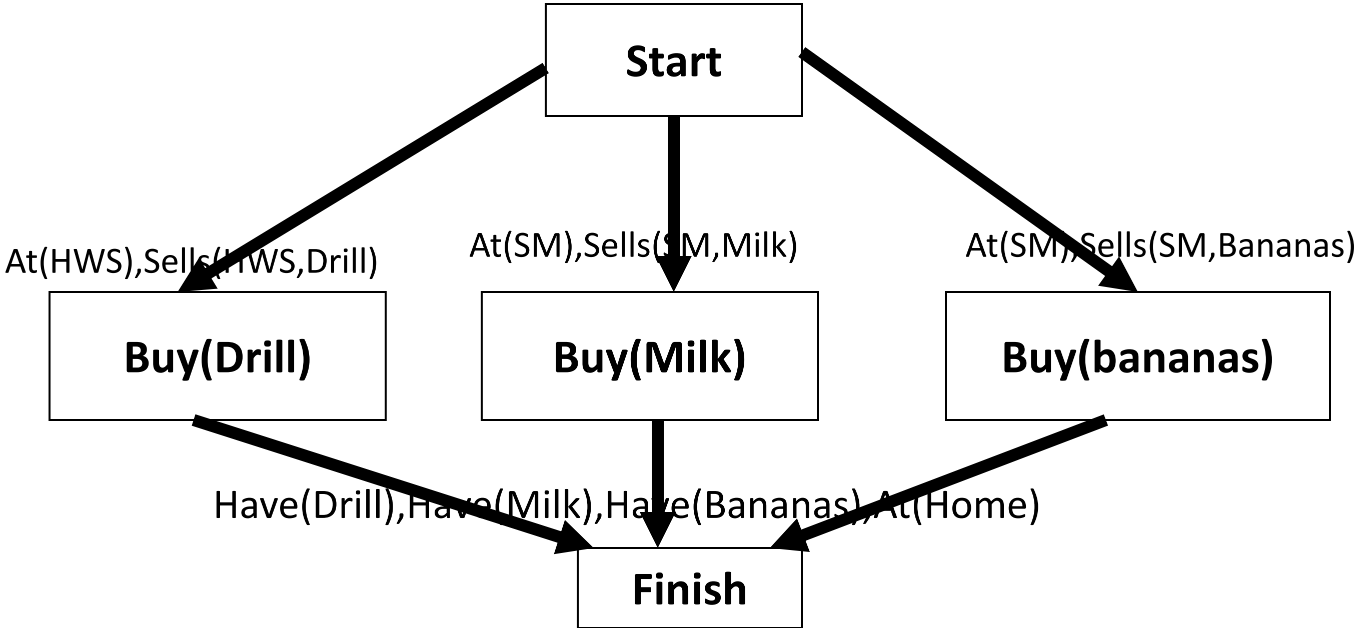
Buy(Drill)

Buy(Milk)

Buy(bananas)

Have(Drill),Have(Milk),Have(Bananas),At(Home)

Finish



Start

At(HWS),Sells(HWS,Drill)

At(SM),Sells(SM,Milk)

At(SM),Sells(SM,Bananas)

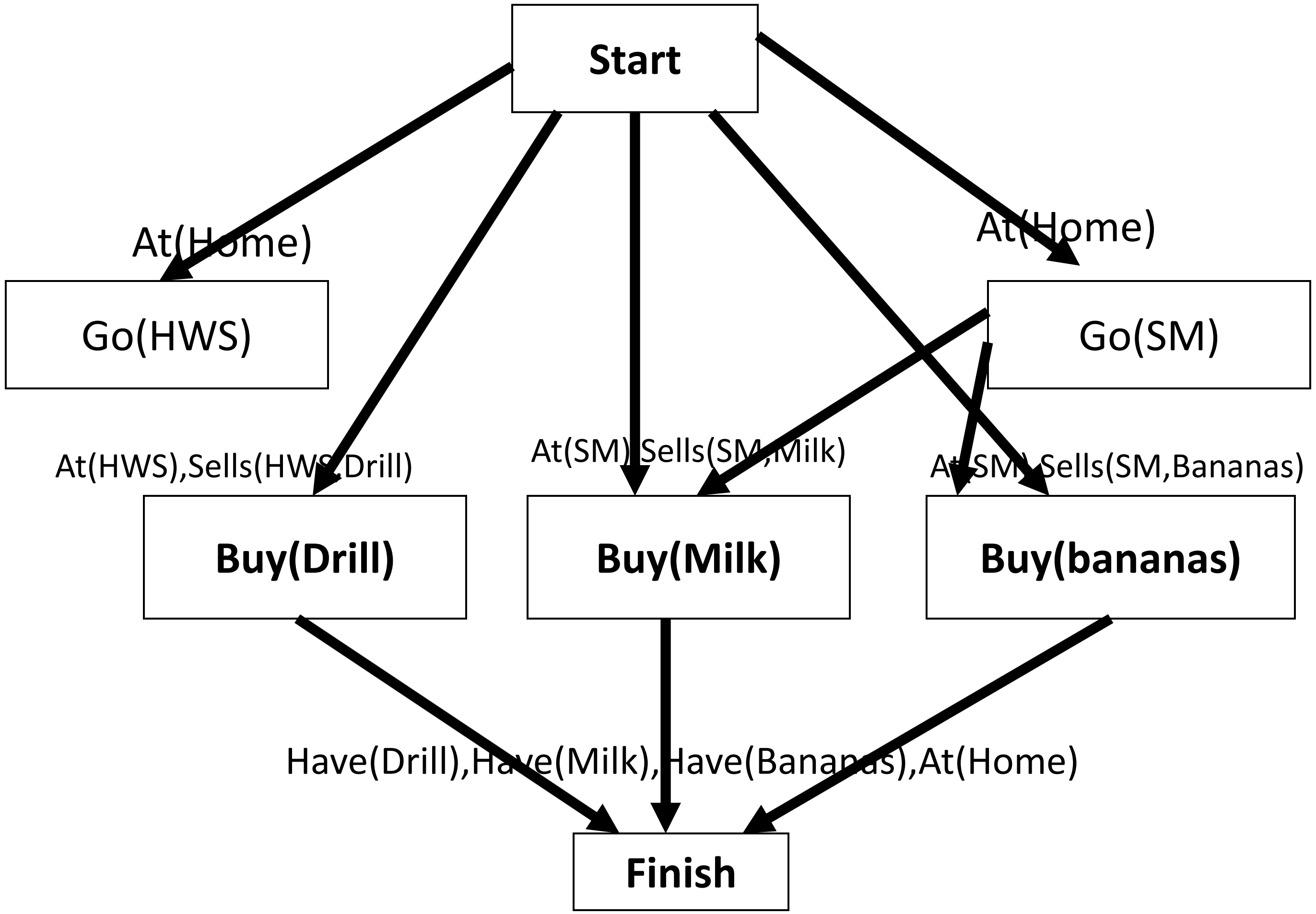
Buy(Drill)

Buy(Milk)

Buy(bananas)

Have(Drill),Have(Milk),Have(Bananas),At(Home)

Finish



Start

At(Home)

Go(HWS)

At(Home)

Go(SM)

At(HWS),Sells(HWS,Drill)

At(SM),Sells(SM,Milk)

At(SM),Sells(SM,Bananas)

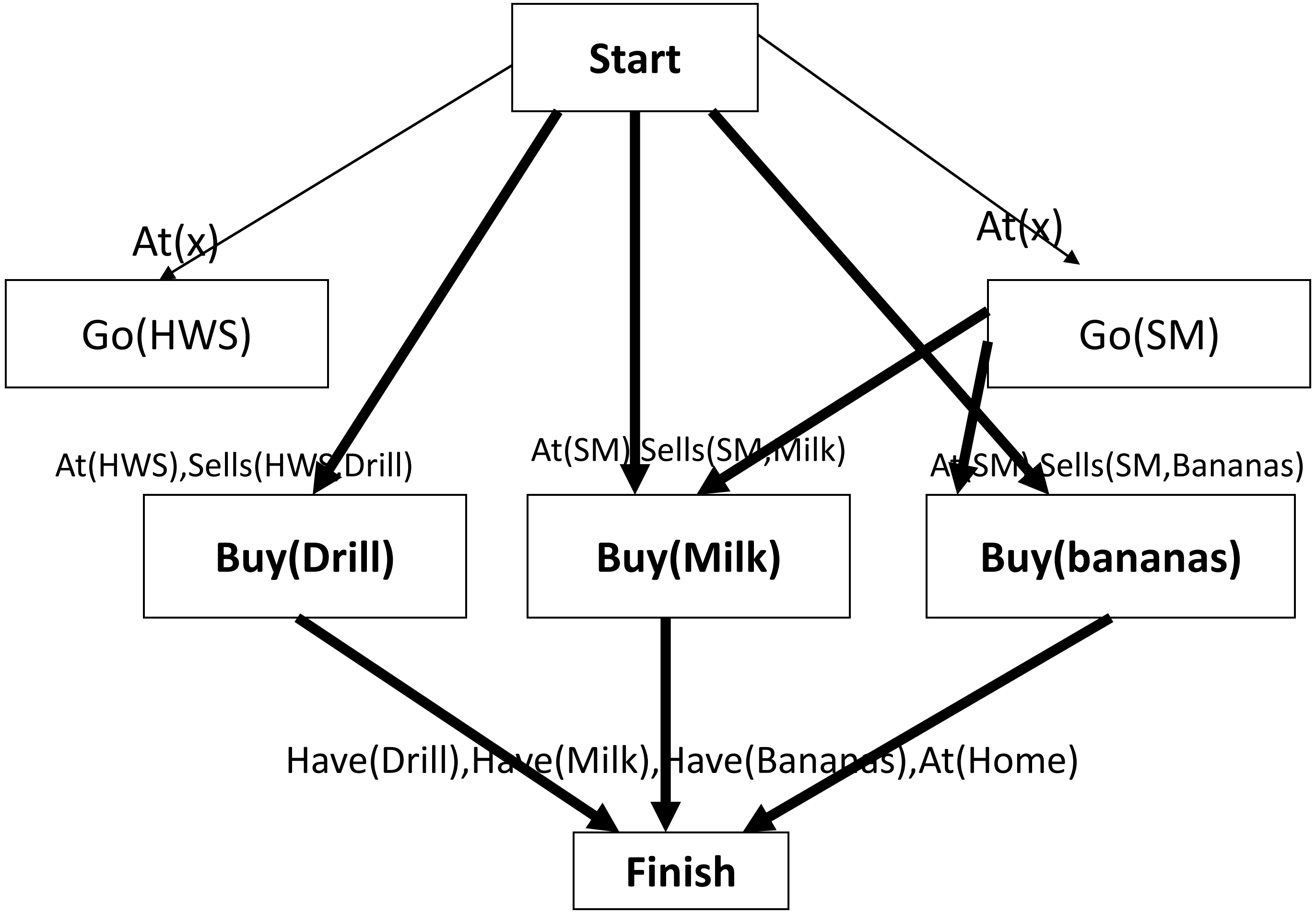
Buy(Drill)

Buy(Milk)

Buy(bananas)

Have(Drill),Have(Milk),Have(Bananas),At(Home)

Finish

Start
At(x)
At(x)
Go(HWS)
Go(SM)
At(HWS),Sells(HWS,Drill)
At(SM),Sells(SM,Milk)
At(SM),Sells(SM,Bananas)
Buy(Drill)
Buy(Milk)
Buy(bananas)
Have(Drill),Have(Milk),Have(Bananas),At(Home)
Finish