VU Einführung in Artificial Intelligence

SS 2024

Hans Tompits

Institut für Logic and Computation
Forschungsbereich Wissensbasierte Systeme

www.kr.tuwien.ac.at

Planning

Planning—General Considerations

Planning = coming up with a sequence of actions that will achieve some goal.

- Reasoning about the results of actions is central to the operation of an intelligent agent.
- One way to represent actions is to use first-order logic expressing things like

  $\forall\, t,$ such-and-such is the result at $t+1$ of doing action at t.

- In what follows, we describe an approach to planning which avoids explicit times and focusses instead on *states*.
  - A state results from another state by applying some action.

Planning—General Considerations (ctd.)

- ➤ In dealing with reasoning about actions, three problems have in this context been identified in the literature:
  - the *frame problem*,
  - the *ramification problem*, and
  - the *qualification problem*.

- ➤ The frame problem deals with the question how to represent things which stay unchanged after performing some action.
  - Indeed, most things stay the same when applying a single action
    - ➥ a large number of so-called *frame axioms* would be needed in general to represent what does not change by performing an action.

Planning—General Considerations (ctd.)

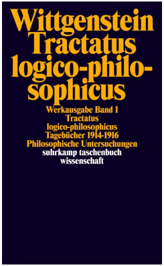
- The ramification problem deals with the representation of *implicit effects*.
  - E.g., if a car moves from one position to another, so does
    - any person in the car, the engine of the car, any dust particle in the car, any bacteria in the driver, etc.
- The qualification problem deals with the required preconditions (the "qualifications") ensuring that an action succeeds.
  - E.g., if a robot needs to move a block A on top of another block B, the following requirements may apply:
    - B should have a clear top, A must not be too heavy, the robot's arm must not be broken, etc.
  - The qualification problem thus deals with a *correct conceptualisation of things*
    - ➥ there is no general solution for it.

Search vs. planning

Applying standard search algorithms for large, real-world planning problems quickly yields enormous search spaces due to irrelevant actions.

- Consider the task of buying a copy of Wittgenstein's *Tractatus logico-philosophicus* from an online bookseller.
- Suppose there is one buying action for each 13-digit ISBN number, hence there are 10^{13} actions in total.

➥ The search algorithm would have to examine the outcome states of all 10^{13} actions to find one satisfying the goal, having a copy of ISBN 9783518281017.



Search vs. planning (ctd.)

- A sensible planning agent, however, should be able to work back from an explicit goal description like $Have(ISBN9783518281017)$.
  - To do this, the agent simply needs the general knowledge that $Buy(x)$ results in $Have(x)$.
  - Given this knowledge and the goal, the planner can decide in a single unification step that $Buy(ISBN9783518281017)$ is the right action.
- The next difficulty is to find a good heuristic function.
  - Suppose the agent's goal is to buy four different books online.
    - ➥ There will be $(10^{13})^4 = 10^{52}$ plans of four steps!
    - ➥ Searching without an accurate heuristic is out of the question!
- Also, the problem solver might be inefficient because it cannot take advantage of *problem decomposition*, which means that it can work on subgoals *independently*.

The Language of Planning Problems

☞ In what follows, we are only concerned with *classical planning* environments, which are

- *fully observable*,
- *deterministic*,
- *finite*,
- *static* (change happens only when the agent acts), and
- *discrete* (in time, actions, objects, and effects).

The Language of Planning Problems (ctd.)

- Key issues of a good planning language:
  - expressive enough to describe a wide variety of problems;
  - restrictive enough to allow efficient algorithms.
- Many different planning languages have been introduced in the literature.
  - ➥ These have been systematised within a standard syntax called the *Planning Domain Definition Language*, or *PDDL* (Ghallab, Howe, Knoblock, McDermott, 1998).
- A base for most of the languages within PDDL is STRIPS (Fikes and Nilsson, 1971), which we discuss in the following.
  - "STRIPS" stands for "Stanford Research Institute Problem Solver".
  - It was designed as the planning component of the software for the Shakey robot project at SRI, which was one of the first major planning systems.

The Language of Planning Problems (ctd.)

Shakey, the Robot (1966-72)

STRIPS—States and Goals

The syntax of STRIPS consists of the following items:

- Representation of states: Planners decompose the world into logical conditions and represent a state as a conjunction of positive literals, referred to as *fluents*.
  - Literals are atomic formulas or negations thereof (a positive literal is just an atom)
    - literals can be propositional or first-order, but first-order literals must be *ground* (i.e., variable-free) and *function-free*.
  - For instance,
    - $Rich \wedge InJail$ may represent the state of some person,
    - while $At(x, y)$ or $At(president(USA), White\_House)$ are not allowed.
  - Furthermore, the *closed-world assumption* is used, meaning that any condition not mentioned in a state is assumed false.

STRIPS—States and Goals (ctd.)

- Representation of goals: A goal is a partially specified state, represented as a conjunction of positive ground literals, such as $Rich \wedge Famous$ or $At(P_2, LakeTahoe)$.
  - A propositional state s *satisfies* a goal g if s contains all the atoms in g (and possibly others).
  - E.g., the state $Rich \wedge Famous \wedge InJail$ satisfies the goal $Rich \wedge Famous$.

STRIPS—Actions

- 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.
  - E.g., an action for flying a plane from one location to another is:
    $Action(Fly(p, from, to),$
    $\quad \textsc{Precond}: At(p, from) \land Plane(p) \land Airport(from) \land Airport(to)$
    $\quad \textsc{Effect}: \neg At(p, from) \land At(p, to))$
  - More precisely:
    - this is actually an example of an *action schema*,
    - representing *a number of different actions* that can be derived by instantiating the variables p, $from$, and to to different constants.

STRIPS—Action Schemata

In general, an action schema consists of three parts:

- ➤ The *action name* and *parameter list*—e.g., $Fly(p, from, to)$.
- ➤ The *precondition*: a conjunction of function-free positive literals stating what 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*: a conjunction of function-free literals describing how the state changes when the action is executed.
  - A positive literal P in the effect is *true* in the state resulting from the action; a negative literal $\neg P$ results in P being *false*.
  - Variables in the effect must also appear in the action's parameter list.
- ☞ Some planning systems divide the effect into the *add list* for positive literals and the *delete list* for negative literals.

STRIPS—Semantics

An action is *applicable* in any state that satisfies the preconditions; otherwise, the action has no effect.

- For a first-order action schema, establishing applicability involves a substitution for the variables in the precondition.
- E.g., suppose the current state is described by

  $$At(P_1, JFK) \wedge At(P_2, SFO) \wedge Plane(P_1) \wedge Plane(P_2) \\ \wedge Airport(JFK) \wedge Airport(SFO).$$

  This state satisfies the precondition

  $$At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$$

  of action schema $Fly(p, from, to)$ with substitution $\{p/P_1, from/JFK, to/SFO\}$.

  - ➥ The concrete action $Fly(P_1, JFK, SFO)$ is applicable.

STRIPS—Semantics (ctd.)

- Starting in a state s, the *result* of executing an applicable action a is a state s' that results from s by
  - adding any positive literal P in the effect of a and
  - removing any P where $\neg P$ appears in the effect of a.
- Thus, for our flight example, after executing $Fly(P_1, JFK, SFO)$, the current state

$$At(P_1, JFK) \land At(P_2, SFO) \land Plane(P_1) \land Plane(P_2) \\ \land Airport(JFK) \land Airport(SFO).$$

becomes

$$At(P_1, SFO) \land At(P_2, SFO) \land Plane(P_1) \land Plane(P_2) \\ \land Airport(JFK) \land Airport(SFO).$$

STRIPS—Semantics (ctd.)

Remarks:

- If a positive effect is already in s it is not added twice, and if a negative effect is not in s, then that part of the effect is ignored.
- The definition of the semantics of STRIPS embodies the so-called *STRIPS assumption*:
  - Every literal not mentioned in the effect remains unchanged.
  - ➥ This is the way STRIPS deals with the frame problem.

STRIPS—Semantics (ctd.)

➤ Finally, a *solution* for a planning problem is an action sequence that, when executed in the initial state, results in a state that satisfies the goal.

☞ Later on, we will allow solutions to be partially ordered sets of actions, provided that every action sequence that respects the partial order is a solution.

The Action Description Language ADL

- A PDDL language more expressive than STRIPS is *ADL* (Pednault, 1986), the *Action Description Language*.
- In ADL, the *Fly* action can, e.g., be written as follows:
  $Action(Fly(p : Plane, from : Airport, to : Airport),$
  $\quad \textsc{Precond}: At(p, from) \land from \neq to$
  $\quad \textsc{Effect}: \neg At(p, from) \land At(p, to))$
- Note:
  - ADL allows *typing*—e.g., the notation $p : Plane$ is an abbreviation for $Plane(p)$.
  - The precondition $from \neq to$ expresses that a flight cannot be made from an airport to itself.
    - ➥ This could not be expressed succinctly in STRIPS!

STRIPS vs. ADL

STRIPS	**ADL**
Only positive literals in states: $Rich \wedge InJail$	Positive and negative literals in states: $\neg Poor \wedge \neg Free$
Closed-World Assumption: Unmentioned literals are false	Open-World Assumption Unmentioned literals are unknown
Effect $P \wedge \neg Q$ means add P and delete Q	Effect $P \wedge \neg Q$ means add P and $\neg Q$ and delete $\neg P$ and Q
Only ground atoms in goals: $Rich \wedge InJail$	Quantified variables in goals: $\exists x \;\, (At(P_1, x) \wedge At(P_2, x))$ is the goal of having P_1 and P_2 in the same place
Goals are conjunctions: $Rich \wedge Famous$	Goals allow conjunction and disjunction: $\neg Poor \wedge (Famous \vee Smart)$
Effects are conjunctions	Conditional effects are allowed: $when\ P : E$ means E is an effect only if P is satisfied
No support for equality	Equality is built in
No support for types	Variables can have types, as in $(p : Plane)$

Remarks

- STRIPS and ADL are adequate for many real-world domains, but they have some significant restrictions.
  - An important one is that *ramifications* of actions cannot be represented in a natural way.
    - Indirect actions, like dust particles moving with airplanes, need to be represented as *direct effects*
    - ➥ it would be more natural if these changes could be *derived* from the location of the plane.
- Also, classical planning systems do not attempt to address the *qualification problem*.

Example: Air Cargo Transport

We describe in pure STRIPS notation the problem of loading and unloading cargo onto and off planes and flying it from place to place.

- We use three actions: $Load$, $Unload$, and Fly.
- The actions affect two predicates:
  - $In(c, p)$: cargo c is inside plane p,
  - $At(x, a)$: object x is at airport a.

Example: Air cargo transport (ctd.)

$Init(At(C_1, SFO) \wedge At(C_2, JFK) \wedge At(P_1, SFO) \wedge At(P_2, JFK) \wedge Cargo(C_1) \wedge$
$Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2) \wedge Airport(SFO) \wedge Airport(JFK))$

$Goal(At(C_1, JFK) \wedge At(C_2, SFO))$

$Action(Load(c, p, a),$
PRECOND: $At(c, a) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a),$
EFFECT: $\neg At(c, a) \wedge In(c, p))$

$Action(Unload(c, p, a),$
PRECOND: $In(c, p) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a),$
EFFECT: $At(c, a) \wedge \neg In(c, p))$

$Action(Fly(p, from, to),$
PRECOND: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to),$
EFFECT: $\neg At(p, from) \wedge At(p, to))$

- The following plan is a solution to the problem:
  $[Load(C_1, P_1, SFO), Fly(P_1, SFO, JFK), Unload(C_1, P_1, JFK),$
  $Load(C_2, P_2, JFK), Fly(P_2, JFK, SFO), Unload(C_2, P_2, SFO)].$

Example: Air Cargo Transport (ctd.)

Init	$At(C_1, SFO) \wedge At(C_2, JFK) \wedge At(P_1, SFO) \wedge At(P_2, JFK) \wedge Cargo(C_1) \wedge$ $Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2) \wedge Airport(SFO) \wedge Airport(JFK))$
$\Downarrow$	$Action(Load(C_1, P_1, SFO),$ PRECOND: $At(C_1, SFO) \wedge At(P_1, SFO) \wedge Cargo(C_1) \wedge Plane(P_1) \wedge Airport(SFO),$ EFFECT: $\neg At(C_1, SFO) \wedge In(C_1, P_1))$
s_1	$In(C_1, P_1) \wedge At(C_2, JFK) \wedge At(P_1, SFO) \wedge At(P_2, JFK) \wedge Cargo(C_1) \wedge$ $Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2) \wedge Airport(SFO) \wedge Airport(JFK))$
$\Downarrow$	$Action(Fly(P_1, SFO, JFK),$ PRECOND: $At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK),$ EFFECT: $\neg At(P_1, SFO) \wedge At(P_1, JFK))$
s_2	$In(C_1, P_1) \wedge At(C_2, JFK) \wedge At(P_1, JFK) \wedge At(P_2, JFK) \wedge Cargo(C_1) \wedge$ $Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2) \wedge Airport(SFO) \wedge Airport(JFK))$
$\Downarrow$	$Action(Unload(C_1, P_1, JFK),$ PRECOND: $In(C_1, P_1) \wedge At(P_1, JFK) \wedge Cargo(C_1) \wedge Plane(P_1) \wedge Airport(JFK),$ EFFECT: $At(C_1, JFK) \wedge \neg In(C_1, P_1))$
s_3	$At(C_1, JFK) \wedge At(C_2, JFK) \wedge At(P_1, JFK) \wedge At(P_2, JFK) \wedge Cargo(C_1) \wedge$ $Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2) \wedge Airport(SFO) \wedge Airport(JFK))$

Example: Air Cargo Transport (ctd.)

s_3	$At(C_1, JFK) \wedge At(C_2, JFK) \wedge At(P_1, JFK) \wedge At(P_2, JFK) \wedge Cargo(C_1) \wedge$ $Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2) \wedge Airport(SFO) \wedge Airport(JFK))$
$\Downarrow$	$Action(Load(C_2, P_2, JFK),$ Precond: $At(C_2, JFK) \wedge At(P_2, JFK) \wedge Cargo(C_2) \wedge Plane(P_2) \wedge Airport(JFK),$ Effect: $\neg At(C_2, JFK) \wedge In(C_2, P_2))$
s_4	$At(C_1, JFK) \wedge In(C_2, P_2) \wedge At(P_1, JFK) \wedge At(P_2, JFK) \wedge Cargo(C_1) \wedge$ $Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2) \wedge Airport(SFO) \wedge Airport(JFK))$
$\Downarrow$	$Action(Fly(P_2, JFK, SFO),$ Precond: $At(P_2, JFK) \wedge Plane(P_2) \wedge Airport(JFK) \wedge Airport(SFO),$ Effect: $\neg At(P_2, JFK) \wedge At(P_2, SFO))$
s_5	$At(C_1, JFK) \wedge In(C_2, P_2) \wedge At(P_1, JFK) \wedge At(P_2, SFO) \wedge Cargo(C_1) \wedge$ $Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2) \wedge Airport(SFO) \wedge Airport(JFK))$
$\Downarrow$	$Action(Unload(C_2, P_2, SFO),$ Precond: $In(C_2, P_2) \wedge At(P_2, SFO) \wedge Cargo(C_2) \wedge Plane(P_2) \wedge Airport(SFO),$ Effect: $At(C_2, SFO) \wedge \neg In(C_2, P_2))$
s_6	$At(C_1, JFK) \wedge At(C_2, SFO) \wedge At(P_1, JFK) \wedge At(P_2, SFO) \wedge Cargo(C_1) \wedge$ $Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2) \wedge Airport(SFO) \wedge Airport(JFK))$

$\Longrightarrow$ s_6 satisfies the goal $At(C_1, JFK) \wedge At(C_2, SFO)$.

Example: Blocks World

One of the most famous planning domains is the *blocks world*
$\Longrightarrow$ consists of a set of cube-shaped blocks sitting on a table.

- The blocks can be stacked, but only one block can fit directly on top of another.
- A robot arm can pick up a block and move it to another position, either on the table or on top of another block.
- The arm can pick up only one block at a time, so it cannot pick up a block that has another one on it.
- The goal is always to build one or more stacks of blocks, specified in terms of what blocks are on top of what other blocks.

Example: Blocks World (ctd.)

- We use $On(b, x)$ to indicate that block b is on x, where x is either another block or the table.
- The action $Move(b, x, y)$ expresses that block b is moved from the top of x to the top of y.
  - One of the preconditions on moving b is that no other block be on it.
  - In ADL, we could state this as a sentence of first-order logic: $\neg\exists x\ \ On(x, b)$, or, equivalently, $\forall x\ \ \neg On(x, b)$.
  - In STRIPS, we use a new predicate, $Clear(x)$, that is true when nothing is on x.

Example: Blocks World (ctd.)

- We can formally describe *Move* in STRIPS as follows:
  $Action(Move(b, x, y),$
  $\quad \textsc{Precond}: On(b, x) \land Clear(b) \land Clear(y),$
  $\quad \textsc{Effect}: On(b, y) \land Clear(x) \land \neg On(b, x) \land \neg Clear(y))$

- But this action does not maintain *Clear* properly when x or y is the table:
  - for $x = Table$, we get $Clear(Table)$, but the table should not become clear,
  - for $y = Table$, it has the precondition $Clear(Table)$, but the table does not have to be clear to move a block onto it.

Example: Blocks World (ctd.)

To fix this, we do the following:

1. We introduce another action to move a block b from x to the table:
   $\mathit{Action}(\mathit{MoveToTable}(b, x),$
   $\quad \textsc{Precond}: \mathit{On}(b, x) \land \mathit{Clear}(b),$
   $\quad \textsc{Effect}: \mathit{On}(b, \mathit{Table}) \land \mathit{Clear}(x) \land \neg \mathit{On}(b, x))$

2. We interpret $\mathit{Clear}(b)$ as "there is a clear space on b to hold a block"
   $\Longrightarrow$ $\mathit{Clear}(\mathit{Table})$ will always be true.

Example: Blocks World (ctd.)

- One caveat in doing this:
  - Nothing prevents a planner from using $Move(b, x, Table)$ instead of $MoveToTable(b, x)$
    - it will lead to a larger-than-necessary search space albeit to no incorrect answers
    - to avoid this, we can introduce the predicate $Block$ and add $Block(b) \wedge Block(y)$ to the precondition of $Move$.
- There is also the problem of spurious actions like $Move(B, C, C)$.
  $\Longrightarrow$ can be avoided by adding inequalities.
- The complete specification of the blocks world problem is given next (in slightly generalised STRIPS notation, as discussed).

Example: Blocks World (ctd.)

$Init(On(A, Table) \wedge On(B, Table) \wedge On(C, Table) \wedge$
$Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(A) \wedge Clear(B) \wedge Clear(C))$

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

$Action(Move(b, x, y),$
PRECOND: $On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge$
$(b \neq x) \wedge (b \neq y) \wedge (x \neq y),$
EFFECT: $On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y))$

$Action(MoveToTable(b, x),$
PRECOND: $On(b, x) \wedge Clear(b) \wedge Block(b) \wedge Block(x) \wedge (b \neq x),$
EFFECT: $On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))$

- The following plan is a solution to the problem:
  $[Move(B, Table, C), Move(A, Table, B)]$.

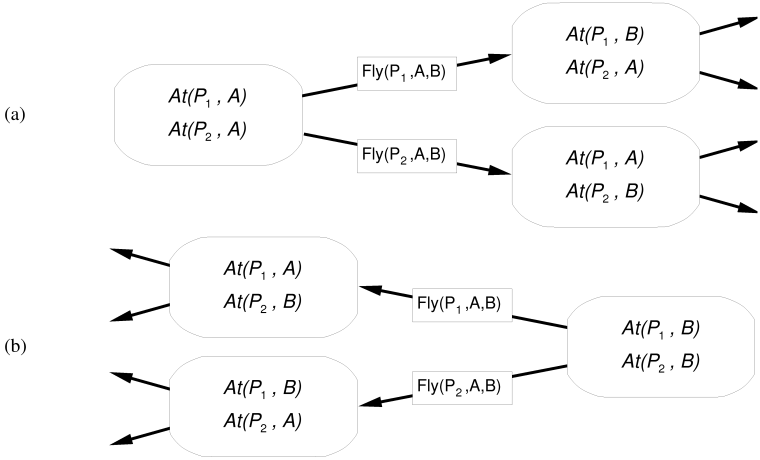
Planning with State-space Search

We now turn to the question of how to *find plans*.

- ➤ The most straightforward approach is to use *state-space search*.
- ➤ Two possibilities:
  - forward state-space search (or *progression planning*): from initial state to goal;
  - backward state-space search (or *regression planning*): from goal to initial state.

Planning with State-space Search (ctd.)

The two approaches illustrated: (a) progression planning; (b) regression planning.



Progression Planning

- We start in the problem's initial state, considering sequences of actions until we find a sequence that reaches a goal state.
- The formulation of planning problems as state-space search problems is as follows:
  - The *initial state* of the search is the initial state of the planning problem.
    - Each state will be a set of positive ground literals;
    - literals not appearing are false.
  - The *actions* that are applicable to a state are all those whose preconditions are satisfied.
    - The successor state resulting from an action is generated by adding the positive effect literals and deleting the negative effect literals.
    - ☞ In case of a first-order logic language, we must apply a unifier from the preconditions to the effect literals.

Progression Planning (ctd.)

- The *goal test* checks whether the state satisfies the goal of the planning problem.
- The *step cost* of each action is typically 1.
  - ☞ Allowing different costs for different actions could be easily realised, but this is seldom done for STRIPS planners.

Note: in the absence of function symbols, the state space of a planning problem is *finite*

- ➥ any complete graph search algorithm (like A^*) yields a complete planning algorithm!

Regression Planning

- ➤ The main advantage of backward search is that it allows to consider only *relevant* actions.

- ☞ An action is relevant to a conjunctive goal if it achieves one of the conjuncts of the goal.

Regression Planning (ctd.)

For instance:

- Consider the cargo problem with 20 pieces of cargo, having the goal

$$At(C_1, B) \wedge At(C_2, B) \wedge \ldots \wedge At(C_{20}, B).$$

- Seeking actions having, e.g., the first conjunct as effect, we find $Unload(C_1, p, B)$ as relevant.
  - This action will work only if its preconditions are satisfied.
    $\Longrightarrow$ any predecessor state must include the preconditions $In(C_1, p) \wedge At(p, B)$.
  - Moreover, the subgoal $At(C_1, B)$ should not be true in the predecessor state.
    $\Longrightarrow$ The predecessor state description is

$$In(C_1, p) \wedge At(p, B) \wedge At(C_2, B) \wedge \ldots \wedge At(C_{20}, B).$$

Regression Planning (ctd.)

Besides insisting that actions *achieve* some desired goal, they should not *undo* any desired literals.

- Actions satisfying this restriction are called *consistent*.
- E.g., $Load(C_2, p, B)$ would not be consistent with the current goal as it would negate the literal $At(C_2, B)$.

Regression Planning (ctd.)

We can now describe the general process of constructing predecessors for backward search.

- Given a goal description G, let A be an action that is relevant and consistent.
- The corresponding predecessor is as follows:
  - Any positive effects of A that appear in G are deleted.
  - Each precondition literal of A is added, unless it already appears.

➥ Any standard search algorithm can be used to carry out the search.

☞ In the first-order case, satisfaction might require a substitution for variables in the predecessor description.

Partial-order Planning

- Forward and backward state-space search are particular forms of *totally ordered plan* searches.
- They explore only *strictly linear sequences* of actions and do not take advantage of problem decomposition.
- ➥ Any planning algorithm that can place two actions into a plan without specifying which comes first is called a *partial-order planner*.

Example

Consider a simple example of putting on a pair of shoes:

$Init()$

$Goal(RightShoeOn \land LeftShoeOn)$

$Action(RightShoe, \textsc{Precond}: RightSockOn, \textsc{Effect}: RightShoeOn)$

$Action(RightSock, \textsc{Effect}: RightSockOn)$

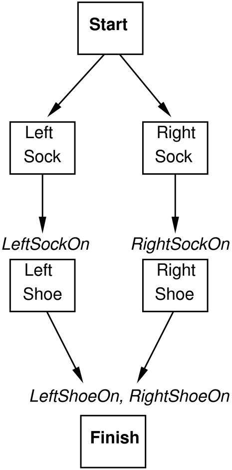
$Action(LeftShoe, \textsc{Precond}: LeftSockOn, \textsc{Effect}: LeftShoeOn)$

$Action(LeftSock, \textsc{Effect}: LeftSockOn)$

- A partial-order planner should come up with the following two-action sequences:
  - $[RightSock, RightShoe]$ to achieve the first conjunct of the goal and
  - $[LeftSock, LeftShoe]$ for the second conjunct.
- Then, the two sequences can be combined to yield the final plan.
- In doing so, the planner manipulates the two subsequences *independently*.

Example (ctd.)

Partial Order Plan:



Start

Left Sock

Right Sock

LeftSockOn

RightSockOn

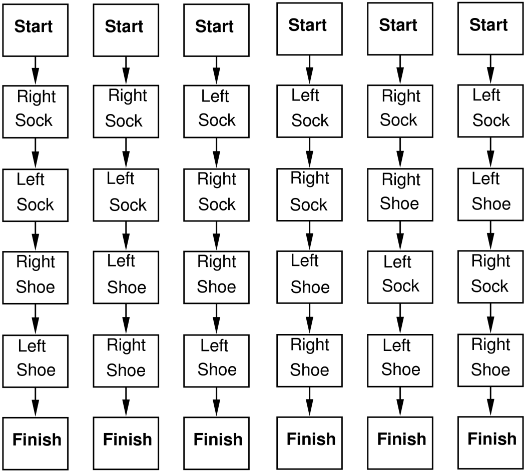
Left Shoe

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—Basics

Partial-order planning can be implemented as a search in the *space of partial-order plans:*

- ➤ We start with an empty plan.
- ➤ Then, we consider ways of refining the plan until we come up with a complete plan that solves the problem.
- ➤ The actions in this search are not actions in the world but *actions on plans:*
  - adding a step to the plan;
  - imposing an ordering that puts one action before another;
  - and so on.

➥ We will define the *POP algorithm* for partial-order planning (as an instance of a search problem).

Partial-order Plans—Components

Each plan has the following four components:

1. a set of *actions*;
2. a set of *ordering constraints*;
3. a set of *causal links*;
4. a set of *open preconditions*.

Partial-order Plans—Components (ctd.)

The set of actions constitutes the elements for making up the steps of the plan.

- The actions 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.

Partial-order Plans—Components (ctd.)

- An ordering constraint is a pair of actions of the form $A \prec B$, read as "A before B".
  - $A \prec B$ 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, like $A \prec B$ and $B \prec A$, represents a *contradiction*
  - ➥ an ordering constraint cannot be added to the plan if it creates a cycle.

Partial-order Plans—Components (ctd.)

- A causal link between two actions A and B in the plan is an expression of form $A \xrightarrow{p} B$, read as "A achieves p for B".
- E.g., the causal link

$$RightSock \xrightarrow{RightSockOn} RightShoe$$

asserts that $RightSockOn$ is an effect of the $RightSock$ action and a precondition of $RightShoe$.

  - It also asserts that $RightSockOn$ must remain true from the time of action $RightSock$ to the time of action $RightShoe$.
  - In other words, the plan may not be extended by adding a new action C that *conflicts* with the causal link.

Partial-order Plans—Components (ctd.)

- An action C *conflicts* with $A \xrightarrow{p} B$ if
  1. C has the effect $\neg p$ and
  2. C could (according to the ordering constraints) come after A and before B.
- 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.

Shoe-and-sock Example Revisited

For instance, the final plan in the shoe-and-sock example has the following components (omitting the ordering constraints that put every other action after *Start* and before *Finish*):

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

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

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

Open preconditions: $\{\}$

Partial-order Plans—Solutions

- We define a *consistent plan* as a plan in which
  - there are no cycles in the ordering constraints and
  - no conflicts with the causal links.
- A *solution* is a consistent plan with no open preconditions.
- Every linearisation of a partial-order solution is a total-order solution whose execution from the initial state will reach a goal state.
- We can extend the notion of "executing a plan" from total-order plans to partial-order plans:
  - A partial-order plan is executed by repeatedly choosing any of the possible next actions.

The POP Algorithm

- The initial plan contains
  - *Start* and *Finish*,
  - the ordering constraint $Start \prec Finish$,
  - no causal links, and
  - all the preconditions in *Finish* as open preconditions.
- The successor function arbitrarily picks
  - one open precondition p on an action B and
  - generates a successor plan for every possible consistent way of choosing an action A that achieves p.

The POP Algorithm (ctd.)

Consistency is enforced as follows:

1. The causal link $A \xrightarrow{p} B$ and the ordering constraint $A \prec B$ are added to the plan.
    - Action A may be an existing action in the plan or a new one.
    - If it is new, add it to the plan and also add $Start \prec A$ and $A \prec Finish$.
2. We resolve conflicts between (i) the new causal link and all existing actions and (ii) action A and all existing causal links, providing A is new.
    - A conflict between $A \xrightarrow{p} B$ and C is resolved by adding $B \prec C$ or $C \prec A$.
    - We add successor states for either or both if they result in consistent plans.

The POP Algorithm (ctd.)

- The goal test checks whether a plan is a solution to the original planning problem.
- ➥ Because only consistent plans are generated, the goal test just needs to check that there are no open preconditions.

Planning—Summary

- Planning systems are problem-solving algorithms that operate on explicit propositional or first-order representations of states and actions.
- The PDDL family of planning languages allow a factored representation of planning problems, containing STRIPS and ADL as particular languages.
- State-space search can operate in the forward direction ("progression") or the backward direction ("regression").
- Partial-order planning algorithms explore the space of plans without committing to a totally ordered sequence of actions.

Planning—Summary (ctd.)

- Different heuristics are defined in the literature to significantly prune the search space.
- A further approach to solve planning problems is by *translating them into formulas of propositional logic* such that
  - the *plans* of a given planning problem P are given by the *models* of the associated formula A.

☞ This method is referred to as *planning as satisfiability*.