
INF5390 - Kunstig intelligens

Agents That Plan

Roar Fjellheim

Outline

- Planning agents
- Plan representation
- State-space search
- Planning graphs
- GRAPHPLAN algorithm
- Partial-order planning
- Summary

AIMA Chapter 10: Classical Planning

What is planning?

Planning is a type of **problem solving** in which the agent uses **beliefs** about **actions** and their **consequences** to find a **solution plan**, where a plan is a **sequence of actions** that leads from an **initial state** to a **goal state**



Previously described approaches

- Planning by search (INF5390-03)
 - ✓ Atomic representations of states
 - ✓ Very large number of possible actions
 - ✓ Needs good domain heuristics to bound search space
- Planning by logical reasoning (INF5390-04)
 - ✓ Hybrid agent can use domain-independent heuristics
 - ✓ But relies on propositional inference (no variables)
 - ✓ Model size rises sharply with problem complexity
- Neither of these approaches scale directly to industrially significant problems

Factored plan representation

- *Factored* representation of:
 - ✓ Initial state
 - ✓ Available actions in a state
 - ✓ Results of applying actions
 - ✓ Goal tests
- Representation language PDDL
 - ✓ Planning Domain Definition Language
 - ✓ Developed from early AI planners, e.g. STRIPS, pioneering robot work at Stanford in early 1970ies
- Used for *classical planning*
 - ✓ Environment is observable, deterministic, finite, static, and discrete

Representation of states and goals

- *States* are represented by conjunctions of function-free ground literals in first-order logic
- Example: $At(Plane_1, Melbourne) \wedge At(Plane_2, Sydney)$
- Closed-world assumption: Any condition not mentioned in a state is assumed to be false
- Goal state - a partially specified state, *satisfied* by any state that contains the goal conditions
- Example goal: $At(Plane_2, Tahiti)$

Representation of actions

- An *action schema* has three components
 - √ *Action* description: Name and parameters (universally quantified variables)
 - √ *Precondition*: Conjunction of positive literals stating what must be true before action application
 - √ *Effect*: Conjunction of positive or negative literals stating how situation changes with operator application
- Example
 - √ **Action**(*Fly*(p , *from*, *to*),
PRECOND: $At(p, from) \wedge Plane(p) \wedge$
 $Airport(from) \wedge Airport(to)$,
EFFECT: $\neg At(p, from) \wedge At(p, to)$)

How are planning actions applied?

- Actions are *applicable* in states that satisfy its preconditions (by binding variables)
 - ✓ State: $At(P_1, JFK) \wedge At(P_2, SFO) \wedge Plane(P_1) \wedge Plane(P_2) \wedge Airport(JFK) \wedge Airport(SFO)$
 - ✓ Precondition: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
 - ✓ Binding: $\{p/P_1, from/JFK, to/SFO\}$
- State after executing action is same as before, except positive effects added (*add list*) and negative deleted (*delete list*)
 - ✓ New state: $At(P_1, SFO) \wedge At(P_2, SFO) \wedge Plane(P_1) \wedge Plane(P_2) \wedge Airport(JFK) \wedge Airport(SFO)$

Planning solution

- The planned actions that will take the agent from the initial state to the goal state
- Simple version:
 - ✓ *An action sequence*, such that when executed from the initial state, results in a final state that satisfies the goal
- More complex cases:
 - ✓ *Partially ordered set of actions*, such that every action sequence that respects the partial order is a solution

Example - Air cargo planning in PDDL

- **Init**($At(C1, SFO) \wedge At(C2, JFK) \wedge At(P1, SFO) \wedge At(P2, JFK) \wedge Cargo(C1) \wedge Cargo(C2) \wedge Plane(P1) \wedge Plane(P2) \wedge Airport(JFK) \wedge Airport(SFO)$)
- **Goal**($At(C1, JFK) \wedge At(C2, SFO)$)
- **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: $\neg At(p, from) \wedge At(p, to)$)

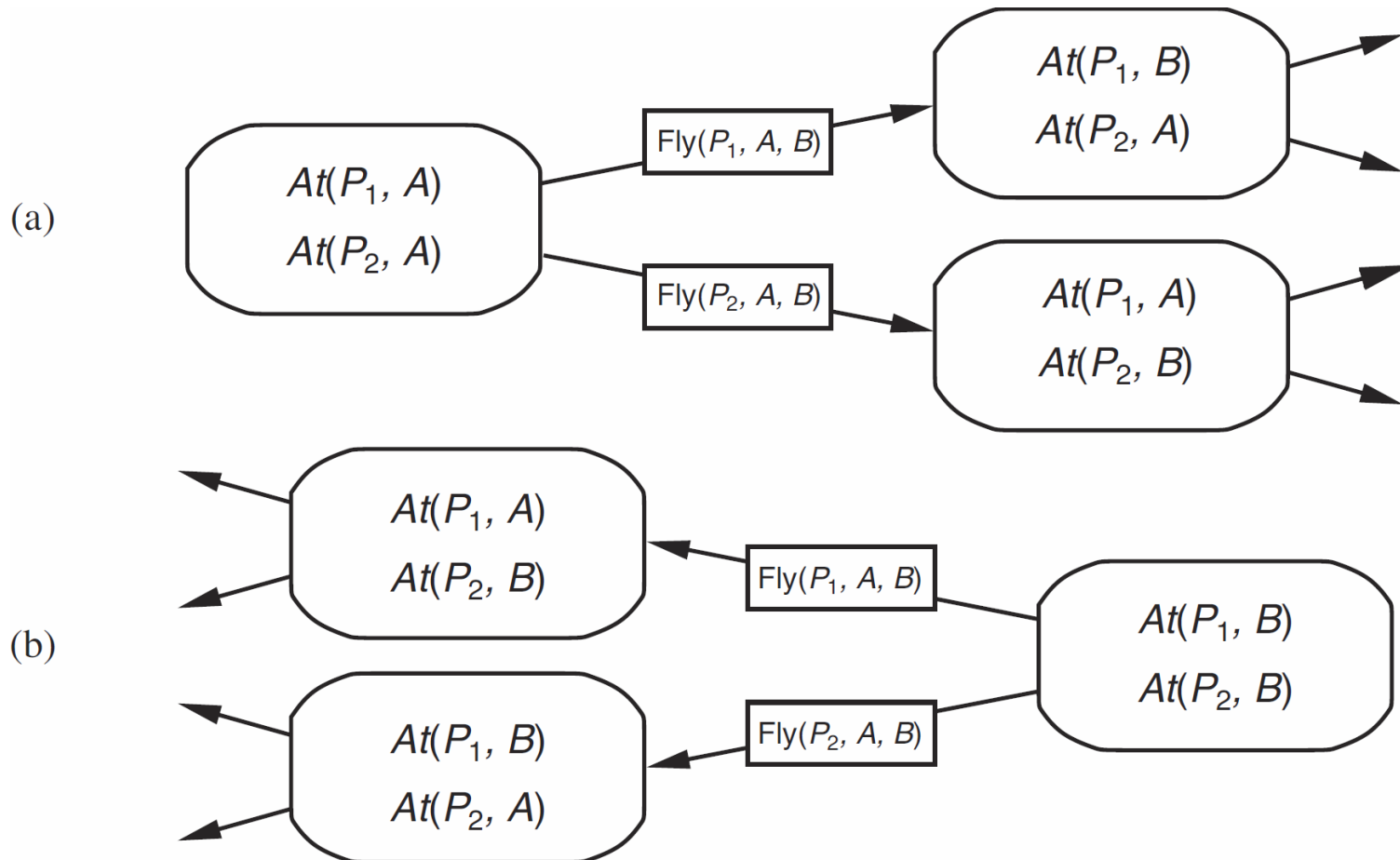
Example – Air cargo solution

- From initial state
 - ✓ **Init**($At(C1, SFO) \wedge At(C2, JFK) \wedge At(P1, SFO) \wedge At(P2, JFK) \wedge Cargo(C1) \wedge Cargo(C2) \wedge Plane(P1) \wedge Plane(P2) \wedge Airport(JFK) \wedge Airport(SFO)$)
- To goal state:
 - ✓ **Goal**($At(C1, JFK) \wedge At(C2, SFO)$)
- Solution – a sequence of actions:
 - ✓ [$Load(C1, P1, SFO), Fly(P1, SFO, JFK), Unload(C1, P1, JFK), Load(C2, P2, JFK), Fly(P2, JFK, SFO), Unload(C2, P2, SFO)$]
- How can the planner generate the plan?

Current popular planning approaches

- Forward state-space search with strong heuristics
 - Planning graphs and GRAPHPLAN algorithm
 - Partial order planning in plan space
 - Planning as Boolean satisfiability (SAT)
 - Planning as first-order deduction
 - Planning as constraint-satisfaction
-
- We will consider the three first ones

Forward and backward state search



Forward state-space search

- *Progression* planning:
 - ✓ Start in initial state
 - ✓ Apply actions whose preconditions are satisfied
 - ✓ Generate successor states by adding/deleting literals
 - ✓ Check if successor state satisfies goal test
- Can be highly inefficient
 - ✓ All actions are applied, even when irrelevant
 - ✓ Large branching factor (many possible actions)
- Heuristics to guide search are required!

Backward state-space search

- *Regression* planning:
 - ✓ Start in goal state
 - ✓ Apply actions that are relevant and consistent
 - Relevant: The action can lead to the goal (adds goal literal)
 - Consistent: The action does not undo (delete) a goal literal
 - ✓ Create predecessor states
 - ✓ Continue until initial state is satisfied
- More efficient, but still requires heuristics
- State-space searches can only produce linear plans

Heuristics for planning

- Neither forward nor backward search is efficient without a good heuristic, which has to be *admissible* (i.e. optimistic)
- Possible heuristics include:
 - ✓ *Adding more edges* to the search graph, thereby making it easier to find a solution path, e.g. ignore pre-conditions or ignore delete lists
 - ✓ *Create state abstractions*, many-to-one mapping from ground states to abstract ones, solve problem in the abstract space, and map down to ground again
- Heuristics generate estimates $h(s)$ for remaining cost of a state that can be used by e.g. A*

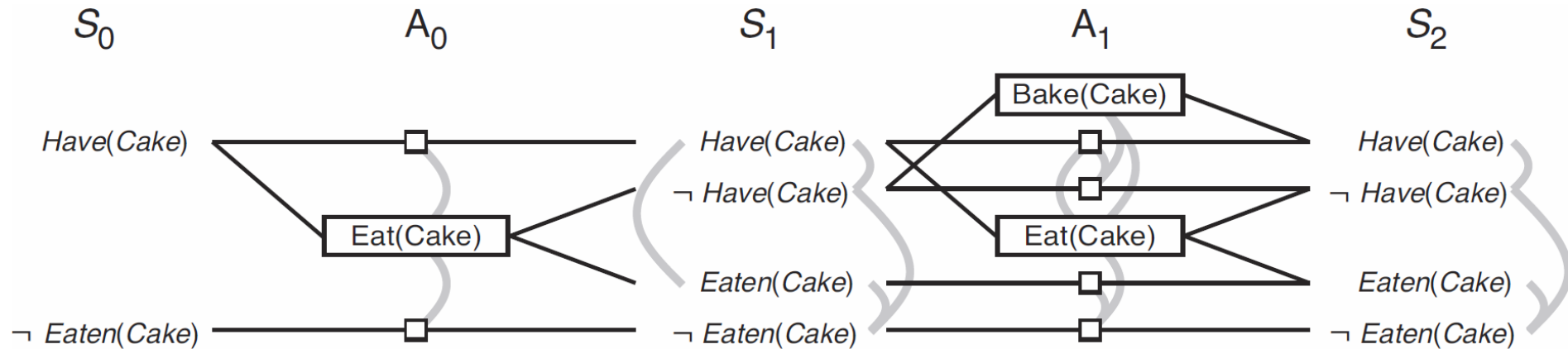
Planning graphs

- A *planning graph* is a special data structure that can be used as a heuristic in search algorithms or directly in an algorithm that generates a solution plan
- Directed graph organized into one *level* for each time step of plan, where a level contains all literals that *may* be true at that step. Literals may be mutually exclusive (*mutex* links)
- Works only for propositional planning problems (no variables), but action schemas with variables may be converted to this form

Example planning problem

- Goal: "Have cake and eat cake too"
- **Init**(*Have(Cake)*)
- **Goal**(*Have(Cake) ∧ Eaten(Cake)*)
- **Action**(*Eat(Cake)*)
PRECOND: *Have(Cake)*
EFFECT: \neg *Have(Cake) ∧ Eaten(Cake)*)
- **Action**(*Bake(Cake)*)
PRECOND: \neg *Have(Cake)*
EFFECT: *Have(Cake)*)

Planning graph for the example



- Alternating state and action layers
- Real and «persistence» actions (small rectangles)
- Mutex links (grey arcs) btw. incompatible states
- Graph *levels off* at S_2 (states repeat themselves)

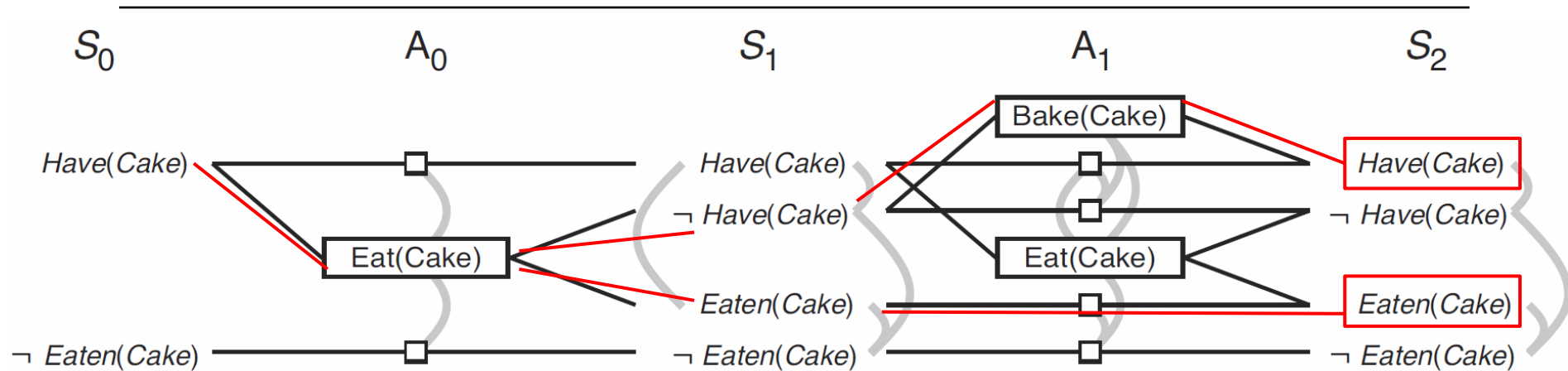
Mutex links (mutual exclusion)

- Between two actions:
 - ✓ *Inconsistent effects* – one action negates an effect of the other (e.g. *Eat(Cake)* and persistent *Have(Cake)*)
 - ✓ *Interference* – an effect of one action negates a pre-condition of the other (e.g. *Eat(Cake)* and *Have(Cake)*)
 - ✓ *Competing needs* – a pre-condition of one action negates a pre-condition of the other (e.g. *Eat(Cake)* and *Bake(Cake)*)
- Between two states (literals):
 - ✓ One literal is the negation of the other
 - ✓ Each possible pair of actions that could achieve the two literals is mutually exclusive

The GRAPHPLAN algorithm

- Uses a planning graph to extract a solution to a planning problem
- Repeatedly
 - ✓ Extend planning graph by one level
 - ✓ If all goal literals are included non-*mutex* in level
 - Try to extract solution that does not violate any *mutex* links, by following links backward in graph
 - Return solution if successful extraction
 - ✓ If the graph has leveled off then report failure
- Creating planning graph is only of polynomial complexity, but plan extraction is exponential

Extracting a solution

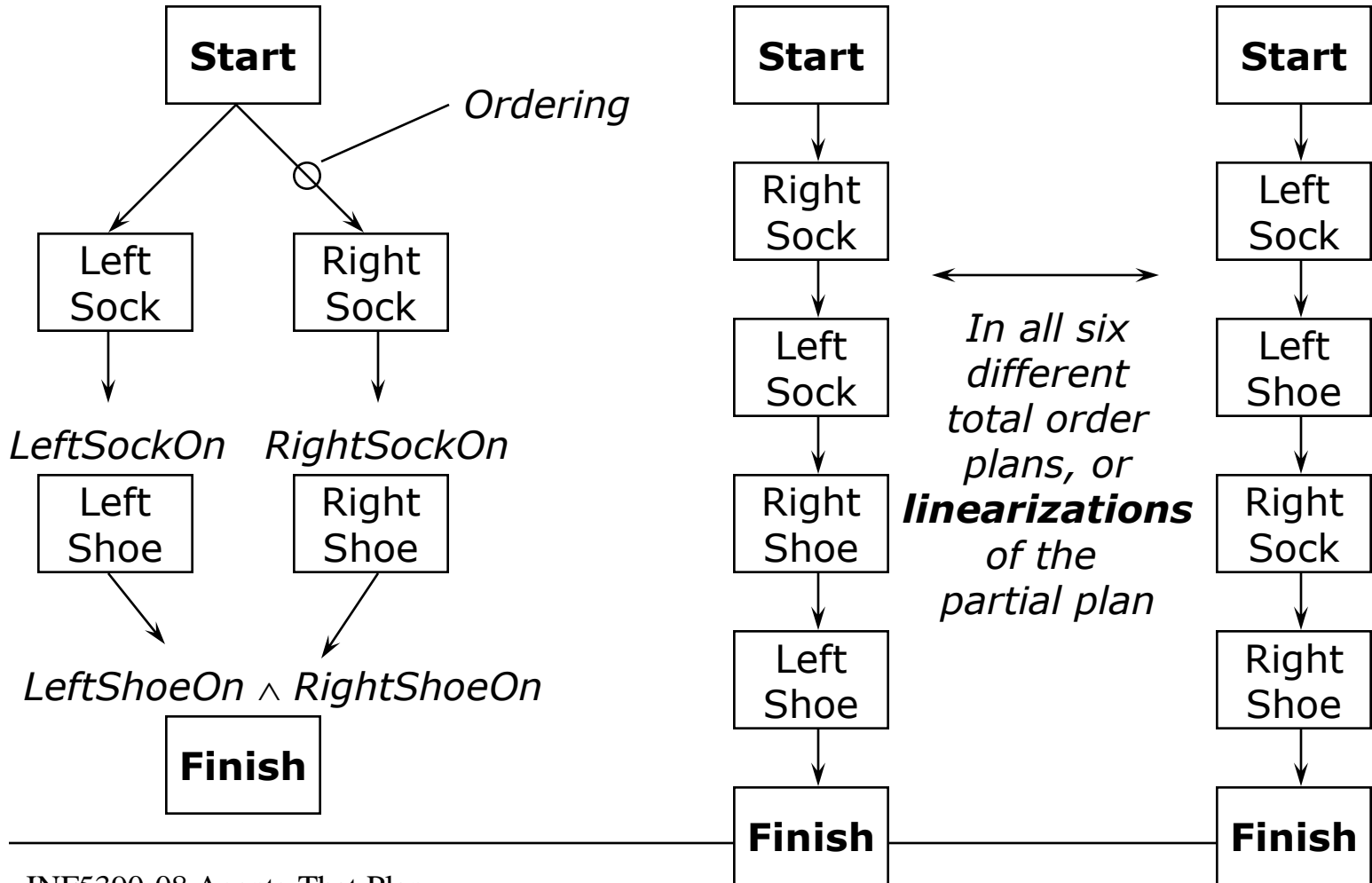


- The goal is $Have(Cake) \wedge Eaten(Cake)$
- Both goal literals non-mutex in S₂
- $Bake(Cake)$ and $Eaten(Cake)$ non-mutex in A₁
- $\neg Have(Cake)$ and $Eaten(Cake)$ non-mutex in S₁
- $Eat(Cake)$ non-mutex in A₀
- $Have(Cake)$ in S₀ is initial state

Partial order planning in plan space

- Each node in the search space corresponds to a (partial) plan
- Search starts with empty plan that is expanded progressively until complete plan is found
- Search operators work in plan space, e.g. *add step*, *add ordering*, etc.
- The solution is the final plan, the path to it is irrelevant
- Can create *partially ordered plans*

Example - Partial and total order plans

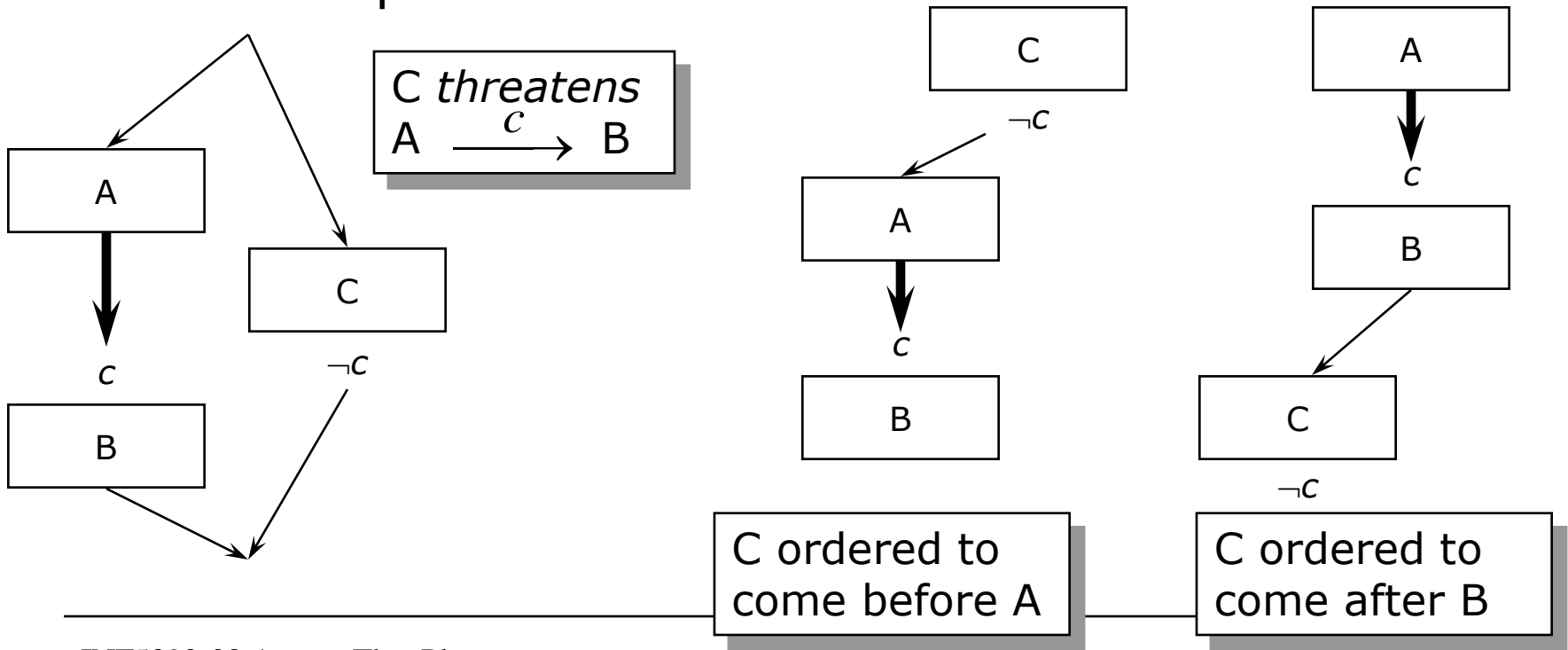


Partial-order plan representation

- A set of *steps*, where each step is an *action* (taken from action set of planning problem)
- Initial empty plan contains just *Start* (no precondition, initial state as effect) and *Finish* (goal as precondition, no effects)
- A set of step *ordering constraints* of the form $A < B$ ("A before B"): A must be executed before B
- A set of *causal links* $A \xrightarrow{c} B$, "A achieves c for B": the purpose of A is to achieve precondition c for B; no action is allowed between A and B that negates c
- Set of *open preconditions*, not achieved by any action yet. The planner must reduce this set to empty set

Protected causal links

- Causal links in a partial plan are *protected* by ensuring that *threats* (steps that might delete the protected condition) are *ordered* to come *before* or *after* the protected link



POP – Partial Order Planning

- Start with initial plan
 - ✓ Contains *Start* and *Finish* steps
 - ✓ All preconditions of *Finish* (goals) as open preconditions
 - ✓ The ordering constraint $Start < Finish$, no causal links
- Repeatedly
 - ✓ Pick arbitrarily one open precondition c on an action B
 - ✓ Generate a successor plan for every consistent way of choosing an action A that achieves c
 - ✓ Stop when a solution has been found, i.e. when there are no open preconditions for any action
- Successful solution plan
 - ✓ Complete and consistent plan the agent can execute
 - ✓ May be partial, agent may choose arbitrary linearization

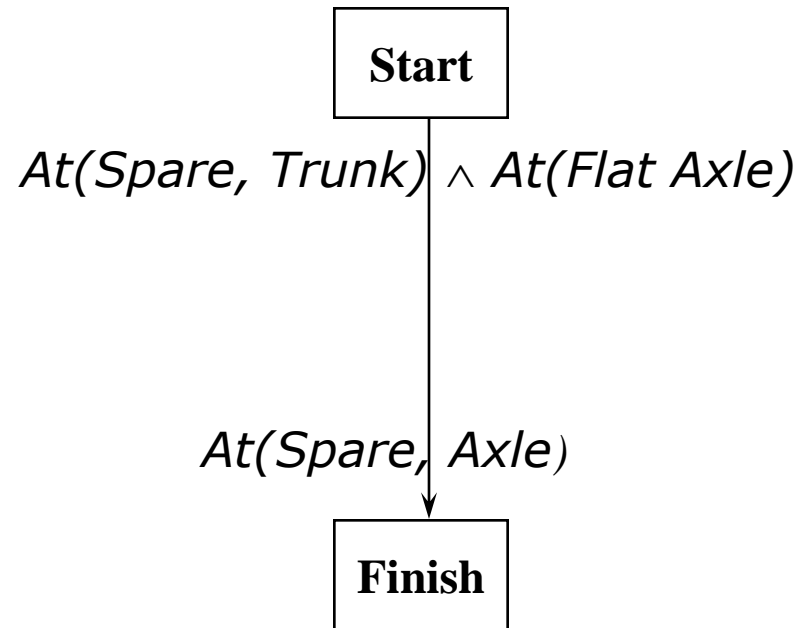
Example – Change tire

- **Init**($At(Flat, Axle) \wedge At(Spare, Trunk)$)
- **Goal**($At(Spare, Axle)$)
- **Action**($Remove(Spare, Trunk)$,
PRECOND: $At(Spare, Trunk)$,
EFFECT: $\neg At(Spare, Trunk) \wedge At(Spare, Ground)$)
- **Action**($Remove(Flat, Axle)$,
PRECOND: $At(Flat, Axle)$,
EFFECT: $\neg At(Flat, Axle) \wedge At(Flat, Ground)$)
- **Action**($PutOn(Spare, Axle)$,
PRECOND: $At(Spare, Ground) \wedge \neg At(Flat, Axle)$,
EFFECT: $\neg At(Spare, Ground) \wedge At(Spare, Axle)$)

Uses ADL language, extends STRIPS

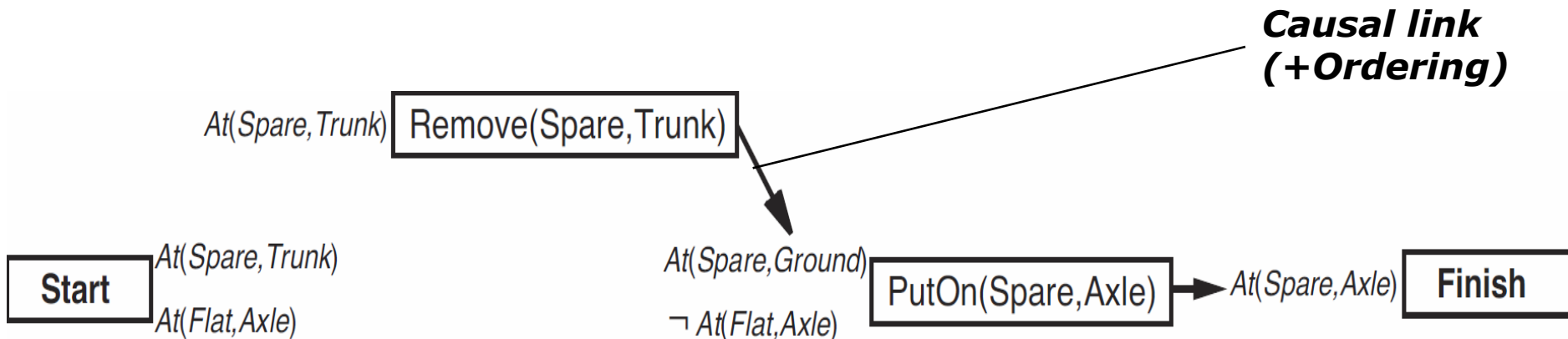
Tire (1) - Initial plan

- For each planning iteration, one step will be added. If this leads to an inconsistent state, the planner will backtrack
- The planner will only consider steps that serve to achieve a precondition that has not yet been achieved



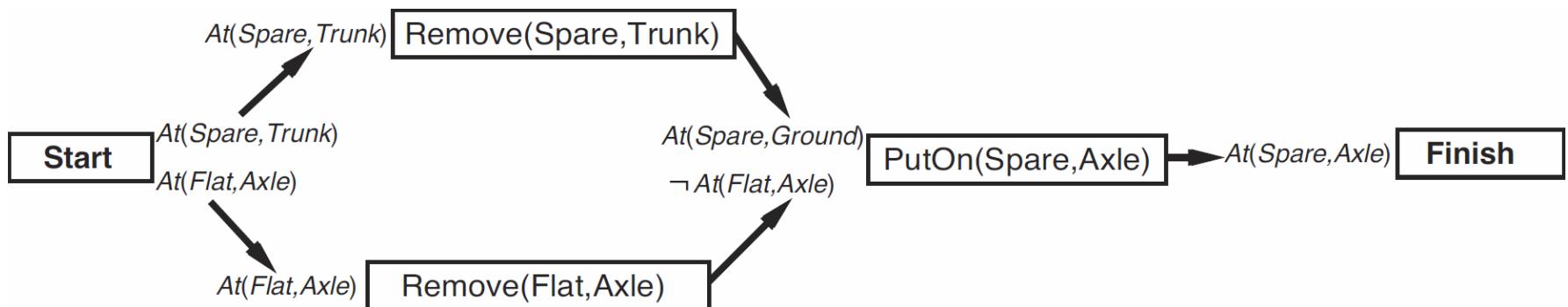
Tire (2) - Achieving open preconditions

- Start by selecting *PutOn* action that achieves *Finish*
- Select *At(Spare, Ground)* precondition of *PutOn*, and choose *Remove(Spare, Trunk)* action
- The planner will *protect* the causal links by not inserting new steps that violate achievements



Tire (3) – Finishing the plan

- Planner selects to achieve $\neg At(Flat, Axle)$ precondition of *PutOn* by *Remove(Flat, Axle)*
- Final two preconditions are satisfied by *Start*



Summary

- Planning agents produce *plans* - sequences of actions - that contribute to reaching goals
- Planning systems operate on *explicit representation* of states, actions, goals, and plans
- *PDDL* (Planning Domain Definition Language) describes *action schemas* in terms of *precondition* and *effects*
- *State-space planning* operates on situations, searches in forward or backward direction, and produces fully ordered plans

Summary (cont.)

- A *planning graph* is a data structure that can be constructed efficiently and be used to extract solution plans (GRAPHPLAN algorithm)
- *Plan-space planning* (POP algorithm) operates on plans, starting with a minimal plan and extending it until a solution is found, and can create partially ordered plans
- Planning is a very active AI field, where techniques are evolving rapidly, and no consensus on best approach exists yet