# Artificial Intelligence

## Lecturer 7 – Part I: Planning

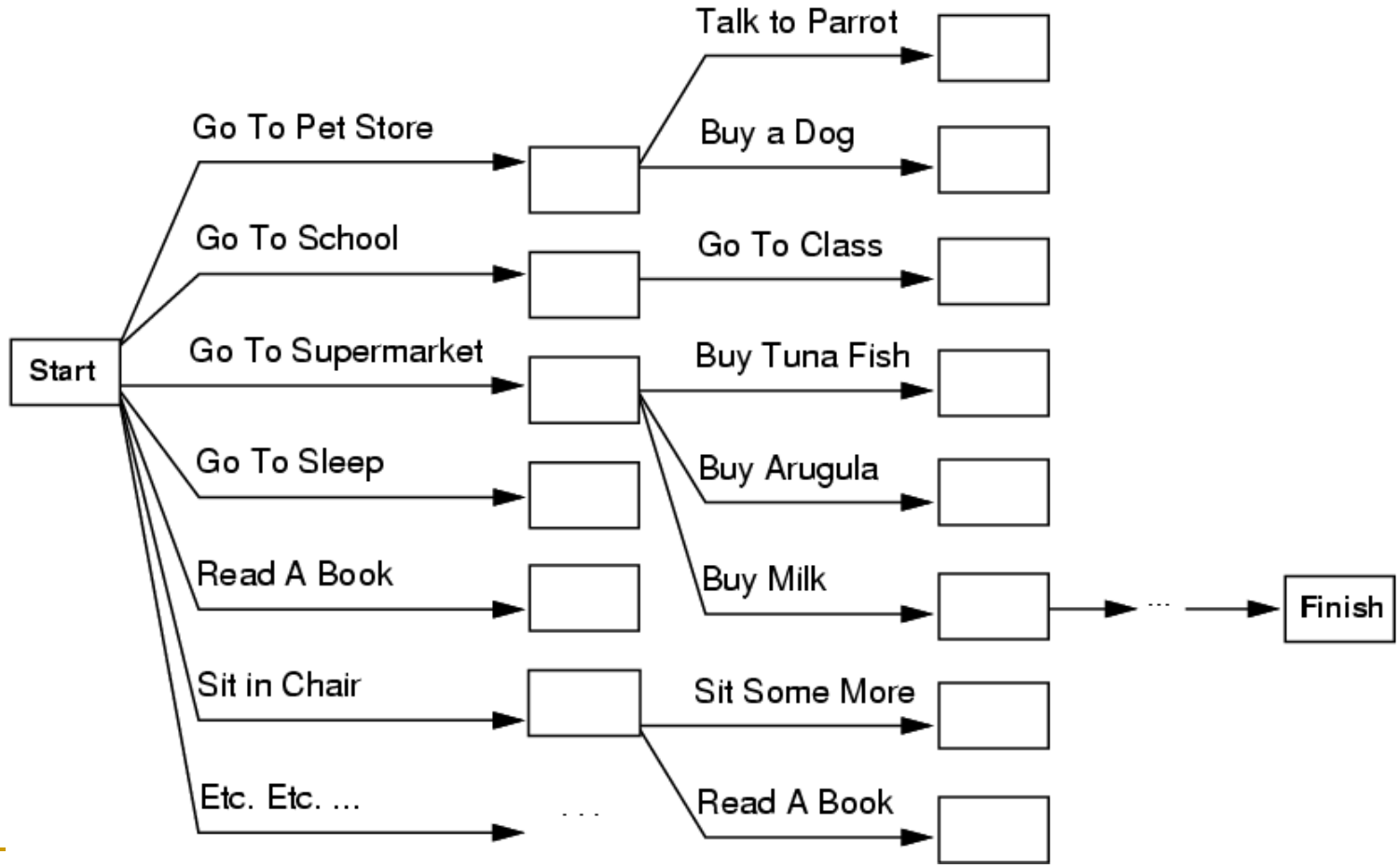
Brigitte Jaumard
Dept of Computer Science and Software Engineering
Concordia University
Montreal (Quebec) Canada

# Outline

- Planning problem
- State-space search
- Partial-order planning
- Planning graphs
- Planning with propositional logic

# Search vs. planning

- Consider the task *get milk, bananas, and a cordless drill*
- Standard search algorithms seem to fail miserably:




- After-the-fact heuristic/goal test inadequate

# Planning problem

- **Planning** is the task of determining a sequence of actions that will achieve a goal.
- **Domain independent heuristics and strategies** must be based on a domain independent representation
  - General planning algorithms require a way to represent states, actions and goals
  - STRIPS, ADL, PDDL are languages based on propositional or first-order logic
- **Classical planning environment**
  - fully observable, deterministic, finite, static and discrete.

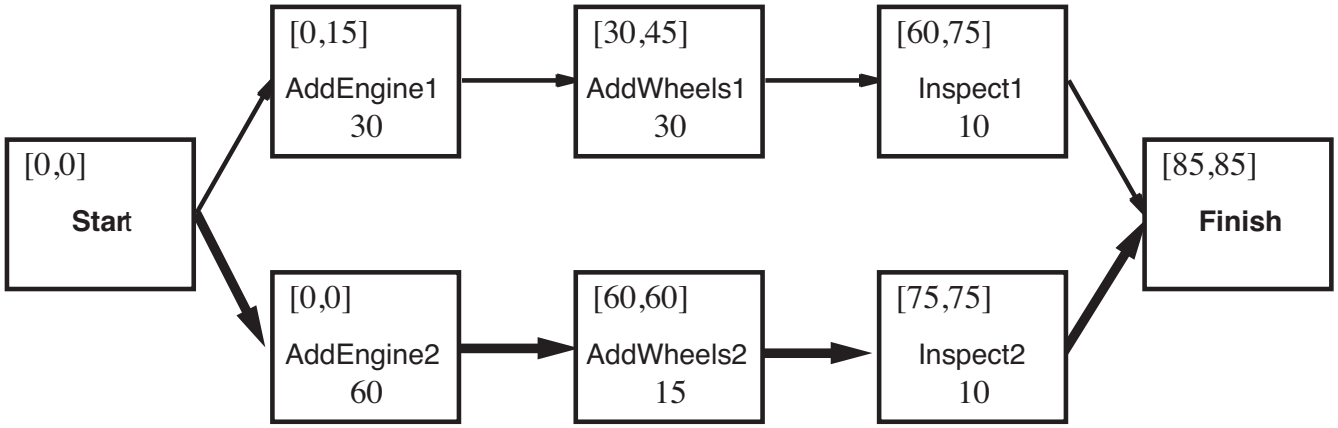
# Additional complexities

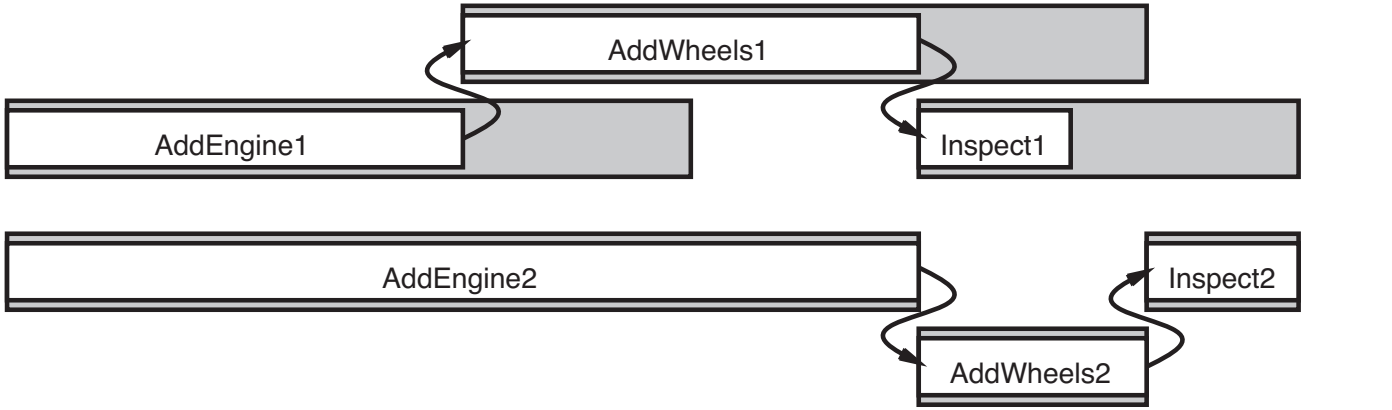
- Because the world is …
  - Dynamic
  - Stochastic
  - Partially observable
- And because actions
  - take time
  - have continuous effects

# AI Planning background

- Focus on classical planning; assume none of the above
- Deterministic, static, fully observable
  - “Basic”
  - Most of the recent progress
  - Ideas often also useful for more complex problems

# CPM: Critical Path Method



# Critical Path Method (Cont'd)

- CPM: path whose duration is the longest → **makespan**
- Actions that are off the critical path have a **time window** in which they can be executed
  - ES: Earliest possible start time
  - LS: Latest possible start time
  - **Slack of an action**: LS – ES
- Complexity of critical path: O(Nb)
  - N: number of actions

# Problem Representation

- State
  - What is true about the (hypothesized) world?
- Goal
  - What must be true in the final state of the world?
- Actions
  - What can be done to change the world?
  - Preconditions and effects

- We'll represent all these as logical predicates

# STRIPS

- Developed at Stanford in early 1970s (Stanford Research Institute Planning System), for the first "intelligent" robot
- *Domain*: a set of typed objects; usually represented as propositions
- *States* are represented as first-order predicates over objects
  - *Closed-world assumption:* everything not stated is false; the only objects in the world are the ones defined
- *Operators/Actions* defined in terms of:
  - *Preconditions:* when can the action be applied?
  - *Effects:* what happens after the action?

  No explicit description of how the action should be executed
- *Goals*: conjunctions of literals

# STRIPS Representation

- States are represented as conjunctions

  In(Robot,room) $\wedge$ $\neg$ In(Charger, r) $\wedge$ ...
- Goals are represented as conjunctions:

  (implicit $\exists$ r) In(Robot, r) $\wedge$ In(Charger, r)
- Actions (operators):
  - Name: Go(here, there)
  - Preconditions: expressed as conjunctions
    At(Robot, here) $\wedge$ Path(here, there)
  - Postconditions (effects): expressed as conjunctions
    At(Robot, there) $\wedge$ $\neg$ At(Robot, here)
- Variables can only be instantiated with objects of the correct type

# STRIPS Operator Representation

- Operators have a name, preconditions and postconditions or effects
- Preconditions are conjunctions of *positive literals*
- Postconditions/effects are represented in terms of:
  - *Add-list*: list of propositions that become true after the action
  - *Delete-list*: list of propositions that become false after the action

# Semantics

- If the precondition is false in a world state, the action does not change anything (since it cannot be applied)
- If the precondition is true:
  - Delete the items in the Delete-list
  - Add the items in the Add-list.

  *Order of operations is important here!*

This is a very restricted language, which means we can do efficient inference.

# Example: Buying Action

- Action: *Buy(x)* (where $x$ is a good)
- Precondition: *At(s)*, *Sells(s,x,p)*, *HaveMoney(p)* (where $s$ is a store, $p$ is the price)
- Effect:
  - Add-list: *Have(x)*
  - Delete-list: *HaveMoney(p)*
- *Note that many important details are abstracted away!*
- Additional propositions can be added to show that now the store has the money, the stock has decreased etc.

# Example: Move Action

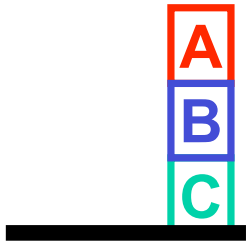
- Action: *Move(object, from, to)*
- Preconditions: *At(object, from), Clear(to), Clear(object)*
- Effects:
  - Add-list: *At(object, to), Clear(from)*
  - Delete-list: *At(object, from), Clear(to)*

# Pros and Cons of STRIPS

- Pros:
  - Since it is restricted, inference can be done efficiently
  - All operators can be viewed as simple deletions and additions of propositions to the knowledge base
- Cons:
  - Assumes that a small number of propositions will change for each action (otherwise operators are hard to write down, and reasoning becomes expensive)
  - Limited language (preconditions and effects are expressed as conjunctions, implicit quantifiers), so not applicable to all domains of interest.

# Example: Block World


*Initial state*



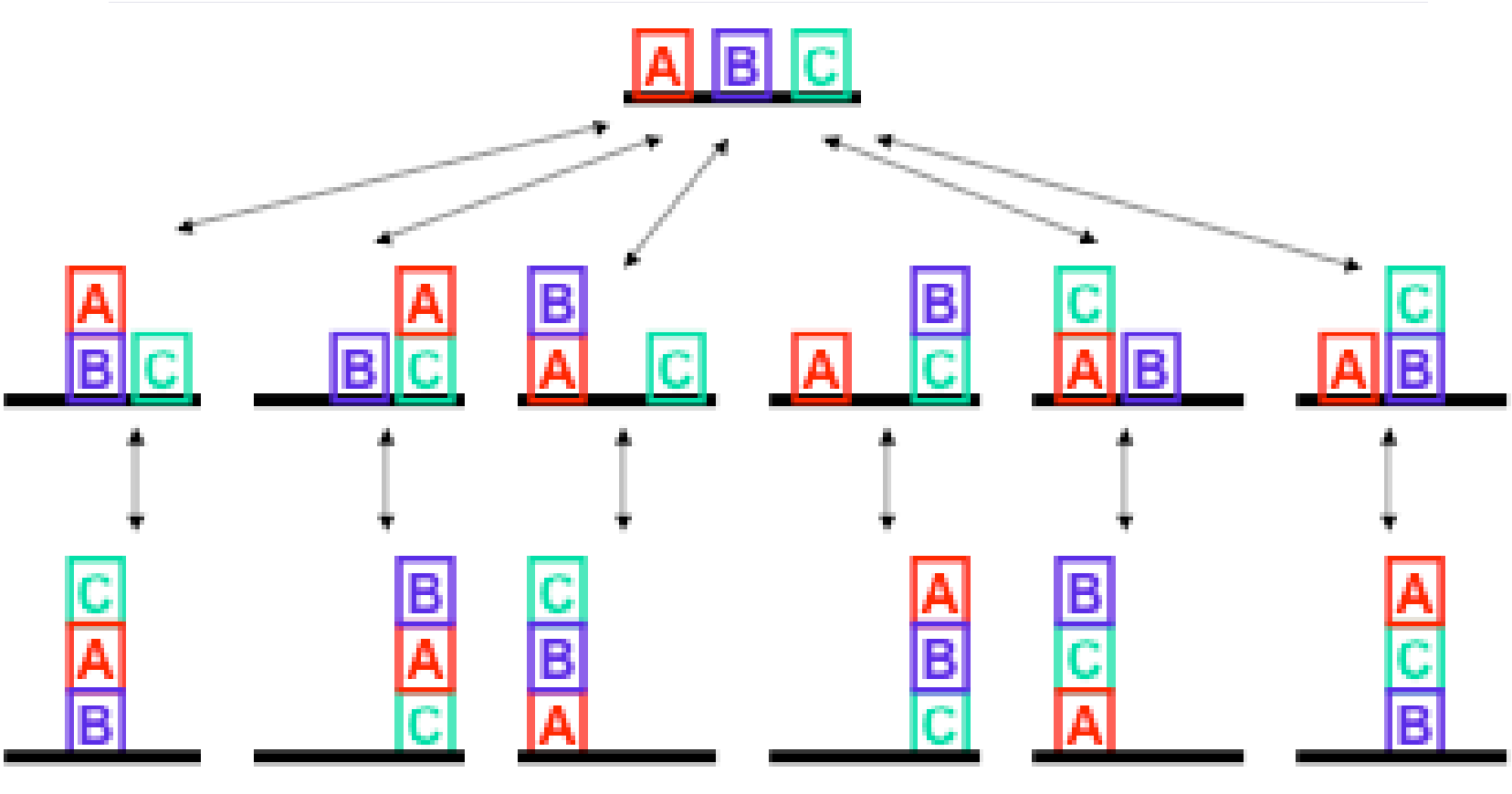

*Goal state*

*Initial state* = On(A,table) ∧ On(B,table) ∧ On(C,table) ∧ Clear(A) ∧ Clear(B) ∧ Clear(C)
*Goal state* = On(A,B) ∧On (B,C)

*Action* = Move(b,x,y)
*Precondition* = On(b,x) ∧ Clear(b) ∧ Clear(y)
*Effect* = On(b,y) ∧ Clear(x) ∧ ¬On(b,x) ∧ ¬Clear(y)

*Action* = MoveToTable(b,x)
*Preconditions* = On(b,x) ∧ Clear(b)
*Effect* = On(b,Table) ∧ Clear(x) ∧ ¬On(b,x)

# State Transitions in the Blocks World



# Two Basic Approaches to Planning

1. *State-space planning* works at the level of the states and operators
   - Finding a plan is formulated as a *search through state space*, looking for a path from the start state to the goal state(s)
   - Most similar to constructive search
2. *Plan-space planning* works at the level of plans
   - Finding a plan is formulated as a *search through the space of plans*
   - We start with a partial, possibly incorrect plan, then apply changes to it to make it a full, correct plan
   - Most similar to iterative improvement/repair

# Planning with state-space search

- Both forward and backward search possible
- Progression planners
  - forward state-space search
  - consider the effect of all possible actions in a given state
- Regression planners
  - backward state-space search
  - Determine what must have been true in the previous state in order to achieve the current state

# Plan-Space Planning in the Blocks World

- Start with plan: *Put(A,B), Put(B,C)*
- Plan fails because the precondition of the second action is not satisfied after the first action
- So we can try to add a step, remove a step, or re-order the steps

# State-Space Planners

- *Progression planners* reason from the start state, trying to find the operators that can be applied (match preconditions)
- *Regression planners* reason from the goal state, trying to find the actions that will lead to the goal (match effects or post-conditions)

In both cases, the planners work with *sets of states* instead of using individual states, like in straightforward search

# Progression (Forward) Planning

1. Determine all operators that are applicable in the start state
2. Ground the operators, by replacing any variables with constants
3. Choose an operator to apply
4. Determine the new content of the knowledge base, based on the operator description
5. Repeat until goal state is reached.

# Example: Supermarket Domain

- In the start state we have *At(Home)*, which allows us to apply operators of the type *Go(x,y)*.
- The operator can be instantiated as *Go(Home, HardwareStore)*, *Go(Home,GroceryStore)*, *Go(Home, School)*, ...
- If we choose to apply *Go(Home, HardwareStore)*, we will delete from the KB *At(Home)* and add *At(HardwareStore)*.
- The new proposition enables new actions, e.g. *Buy*

Note that in this case there are *a lot of possible operators* to perform!

# Goal Regression

- Introduced in Newell & Simon's General Problem Solver
- Algorithm:
  1. Pick an action that satisfies (some of) the goal propositions
  2. Make a new goal by:
     - Removing the goal conditions satisfied by the action
     - Adding the preconditions of this action
     - Keeping any unsolved goal propositions
  3. Repeat until the goal set is satisfied by the start state

# Example: Supermarket Domain

- In the goal state we have $At(Home) \wedge Have(Milk) \wedge Have(Drill)$
- The action *Buy(Milk)* would allow us to achieve *Have(Milk)*
- To apply this action we need to have the precondition *At(GroceryStore)*, so we add it to the set of propositions we want to achieve
- The goal set becomes: $At(Home) \wedge At(GroceryStore) \wedge Have(Drill)$
- Next, we may want to achieve *At(HardwareStore)*

Note that in this case the *order* in which we try to achieve these propositions matters!

# Variations of Goal Regression

- Using a *stack* of goals - also called *linear planning*

  This is *not complete!* I.e. we may not find a plan even if one exists
- Using a *set* of goals - also called *non-linear planning*

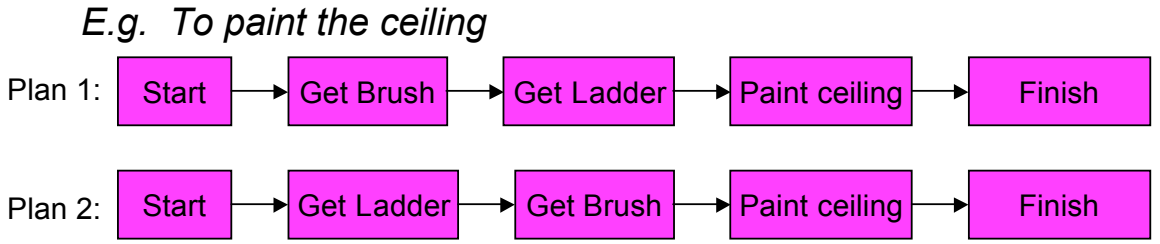
  This is *complete*, but more expensive (need to decide what to work on next)
- Both versions are *sound*: only legal plans will be found

# Prodigy Planner

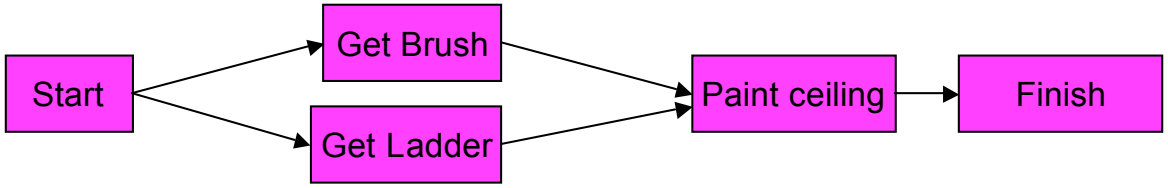
- Do both forward search and goal regression at the same time.
- At each step, choose either an operator to apply or goal to regress
- Uses domain-dependent heuristics to guide the search
- General heuristics (e.g. number of propositions satisfied) do not work well in planning, because subgoals interact

# Total vs. Partial Order

- Total order: Plan is always a strict sequence of actions




- Partial order: Plan steps may be unordered



# Partial Order Planning

- Search in *plan space* and use *least commitment* whenever possible
- In state space search:
  - Search space is a set of states of the world
  - Actions cause transitions between states
  - Plan is a path through state space
- In plan space search:
  - Search space is a set of *partial plans*
  - *Plan operators* cause transitions
  - Goal is a legal plan
- Can maintain both *state* and *plans* (e.g. Prodigy)

# Plan-Space Planners

Plan is defined by $\langle A, O, B, L \rangle$:

- $A$ is a set of actions/operators from the problem domain
- $O$ is a set of ordering constraints of the form $a_i < a_j$

  The constraint specifies that $a_i$ must come before $a_j$ but does not say exactly when
- $B$ is a set of bindings, of the form $v_i = C$, $v_i \neq C$, $v_i = v_j$ or $v_i \neq v_j$, where $v_i, v_j$ are variables and $C$ is a constant
- $L$ is a set of causal links, which records why a certain ordering has to occur:

  $a_i \rightarrow^c a_j$ means that action $a_i$ achieves effect $c$ which is a precondition of $a_j$

# Plan Transformations

- Adding *actions*
- Specifying *orderings*
- *Binding* variables

Constraint satisfaction is used along the way to ensure the consistency of orderings

# Discussion of Partial-Order Planning

- Advantages:
  - Plan steps may be unordered (plan will be ordered, or linearized, before execution)
  - Handles concurrent plans
  - Least commitment can lead to shorter search times
  - Sound and complete
  - Typically produces the optimal plan
- Disadvantages:
  - Complex plan operators lead to high cost for generating every action
  - Larger search space, because of concurrent actions
  - Hard to determine what is true in a state

# The real world

Things are usually not as expected:

- **Incomplete information**
  - Unknown preconditions, e.g., *Intact(Spare)*
  - Disjunctive effects, e.g., *Inflate(x)* causes *Inflated(x)* according to the knowledge base, but in reality it actually causes *Inflated(x)* $\vee$ *SlowHiss(x)* $\vee$ *Burst(x)* $\vee$ *BrokenPump* $\vee \ldots$
- **Incorrect information**
  - Current state incorrect, e.g., spare NOT intact
  - Missing/incorrect postconditions in operators
- **Qualification problem**: can never finish listing all the required preconditions and possible conditional outcomes of actions

# Solutions

- *Conditional (contingency) planning*:
  1. Plans include **observation actions** which obtain information
  2. Sub-plans are created for each contingency (each possible outcome of the observation actions)

  E.g. Check the tire. If it is intact, then we're ok, otherwise there are several possible solutions: inflate, call AAA....

  Expensive because it plans for many unlikely cases
- *Monitoring/Replanning*:
  1. Assume normal states, outcomes
  2. Check progress *during execution*, replan if necessary

  Unanticipated outcomes may lead to failure (e.g., no AAA card)

  In general, some monitoring is unavoidable

# Monitoring

- **Execution monitoring**: "failure" means that the preconditions of the *remaining plan* not met
- **Action monitoring**: "failure" means that the preconditions of the *next action* not met (or action itself fails)

In both cases, need to *replan*

# Replanning

- Simplest: on failure, replan from scratch
- Better: plan to get back on track by reconnecting to best continuation

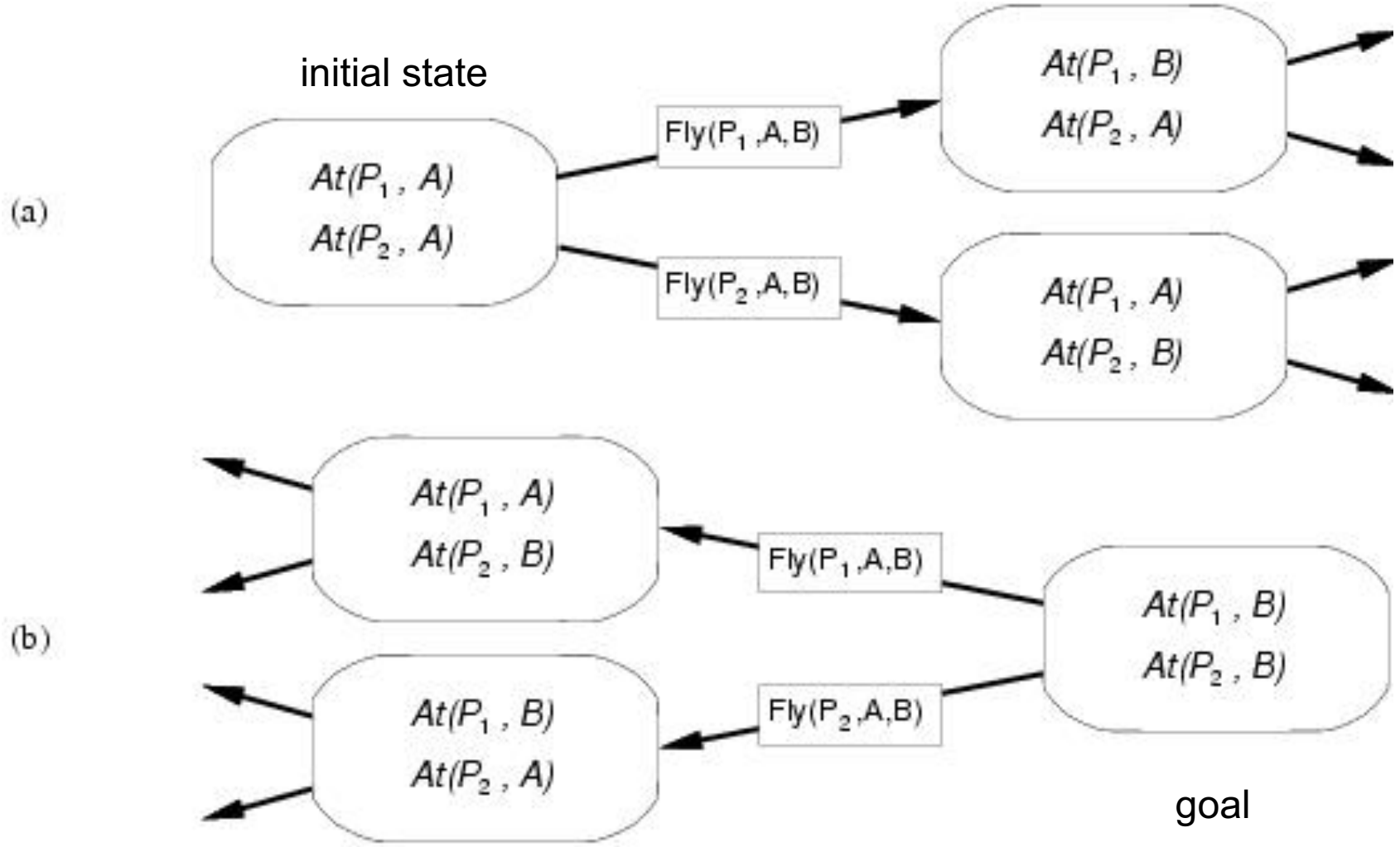
  In this case, we can try to reconnect to the plan's next action, or some future action

  The latter is typically more expensive in terms of planning computation (lots of possible places to reconnect!) but usually yields better plans (e.g. if it is very hard to achieve the preconditions of the very next action)

# Summary

- Planning is very related to search, but allows the actions/states have more structure
- We typically use logical inference to construct solutions
- State-space vs.plan-space planning
- Least-commitment: we build partial plans, order them only as necessary
- In the real world, it is necessary to consider failure cases - replanning
- Hierarchy and abstraction make planning more efficient
- Many varieties of planners that we have not looked at: case-based planners, MDP planners (we will see this later on) etc.

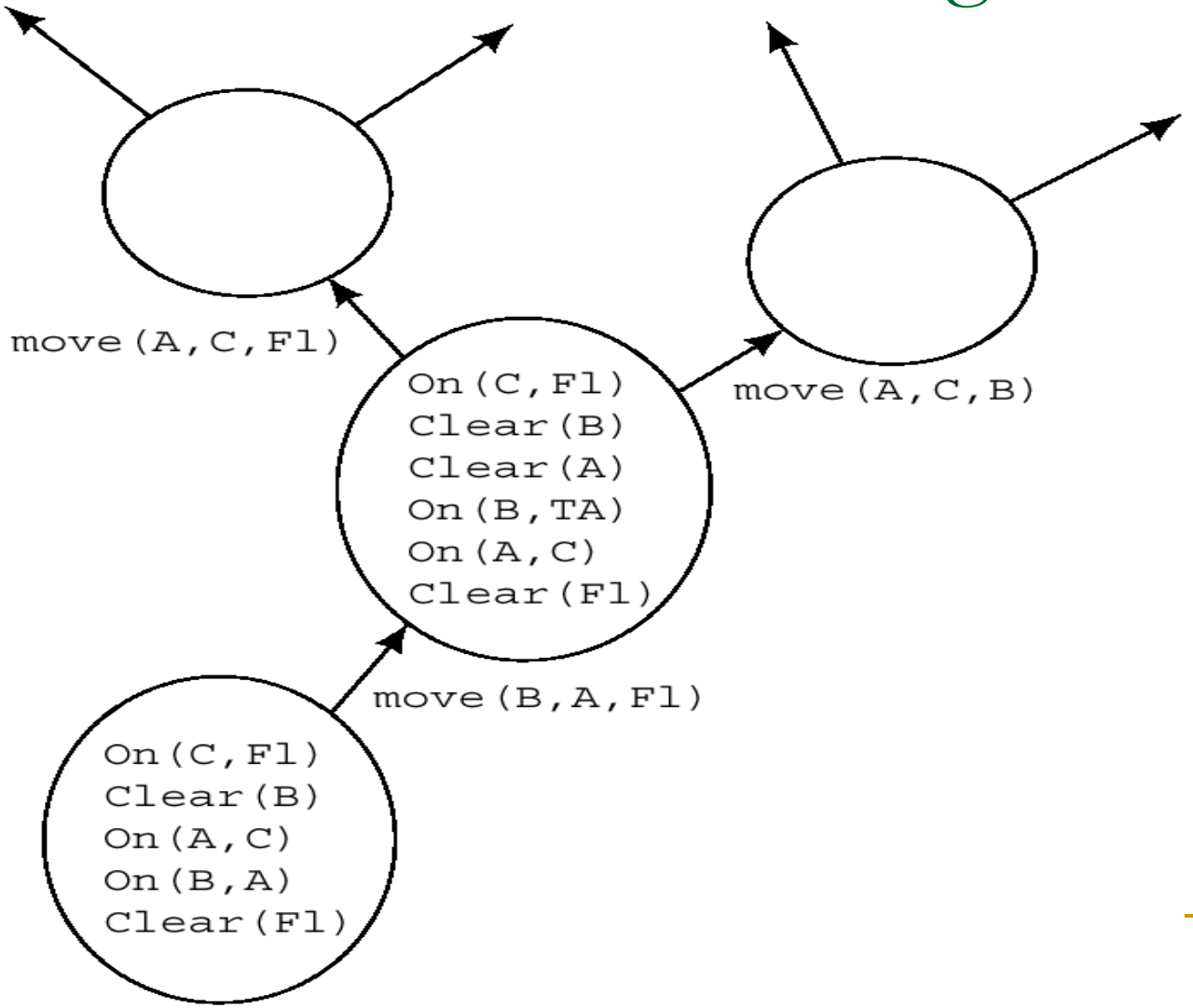
# Progression and regression



# Progression algorithm

- Formulation as state-space search problem:
  - Initial state and goal test: obvious
  - Successor function: generate from applicable actions
  - Step cost = each action costs 1
- Any complete graph search algorithm is a complete planning algorithm.
  - E.g. A*
- Inherently inefficient:
  - (1) irrelevant actions lead to very broad search tree
  - (2) good heuristic required for efficient search

# Forward Search Methods: can use A* with some h and g



# Regression algorithm

- How to determine predecessors?
  - What are the states from which applying a given action leads to the goal?

    Goal state = $At(C1, B) \wedge At(C2, B) \wedge \ldots \wedge At(C20, B)$

    Relevant action for first conjunct: *Unload(C1,p,B)*

    Works only if pre-conditions are satisfied.

    Previous state= $In(C1, p) \wedge At(p, B) \wedge At(C2, B) \wedge \ldots \wedge At(C20, B)$

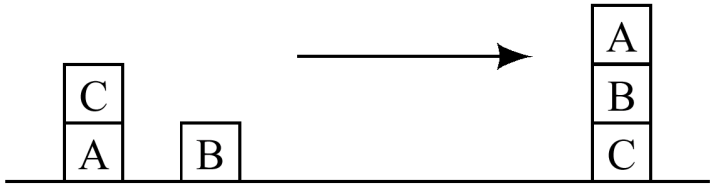
    Subgoal At(C1,B) should not be present in this state.
- Actions must not undo desired literals (consistent)
- Main advantage: only relevant actions are considered.
  - Often much lower branching factor than forward search.

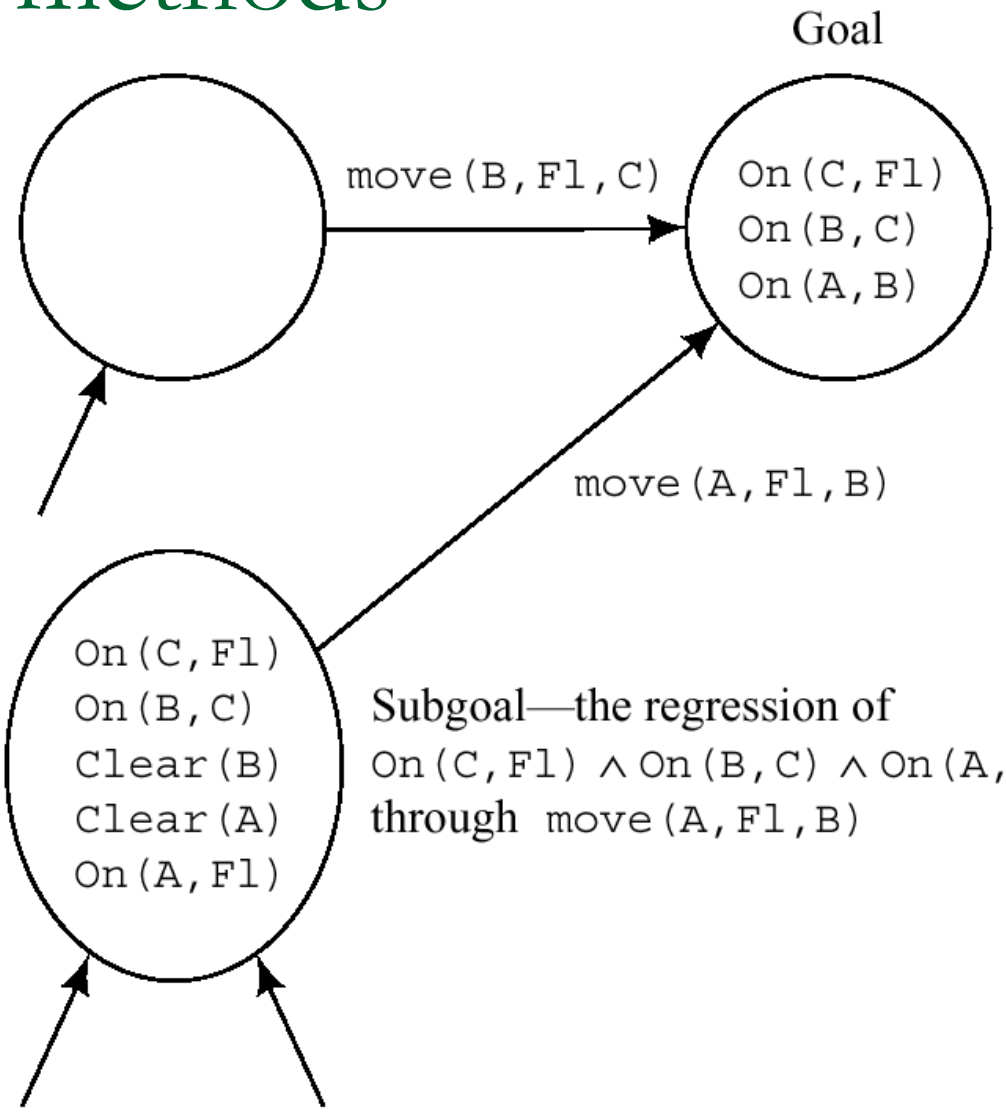
# Regression algorithm

- General process for predecessor construction
  - Give a goal description G
  - Let A be an action that is relevant and consistent
  - The predecessors are 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 added to perform the search.
- Termination when predecessor satisfied by initial state.
  - In FO case, satisfaction might require a substitution.

# Backward search methods

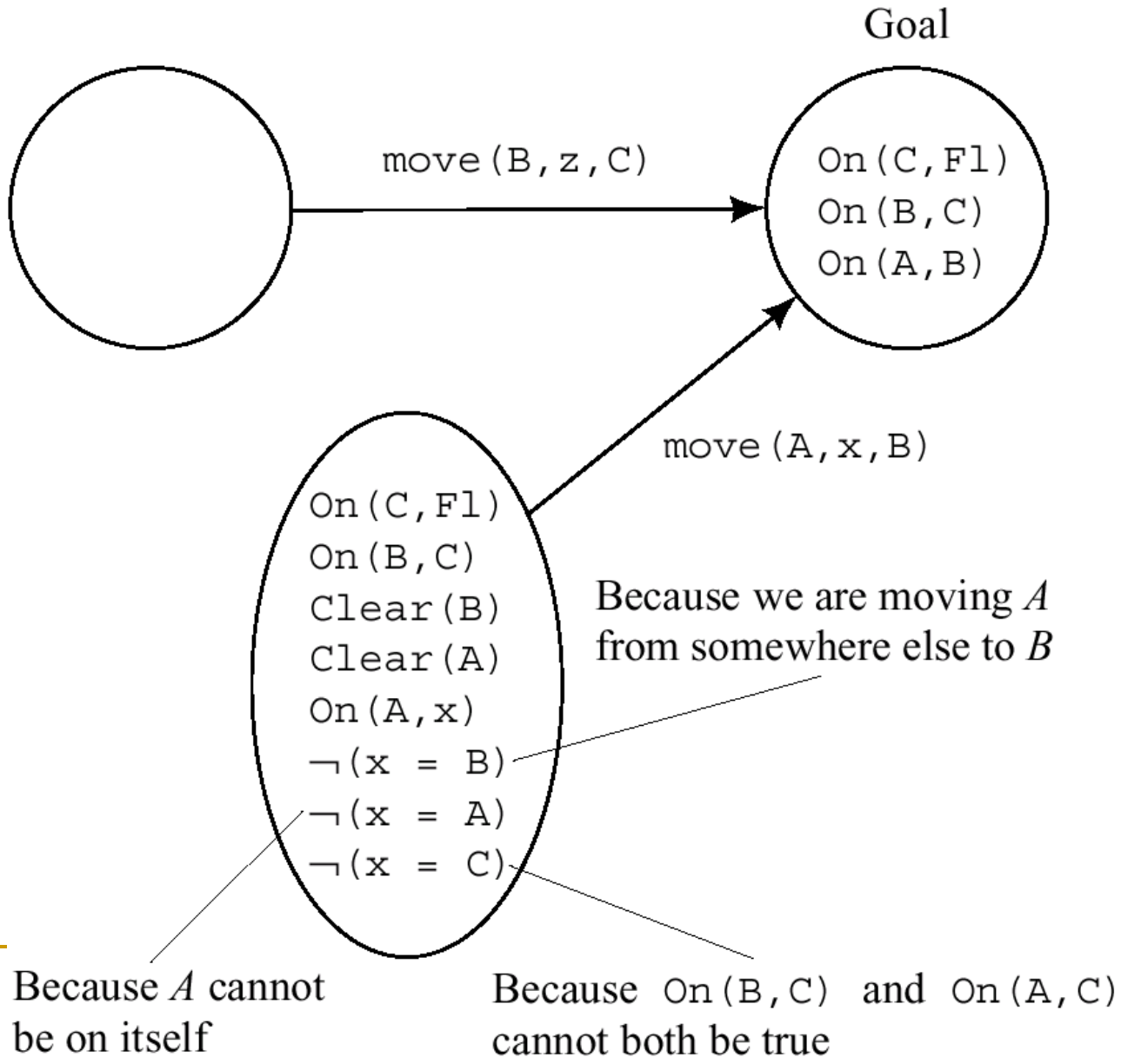
Regressing a ground operator



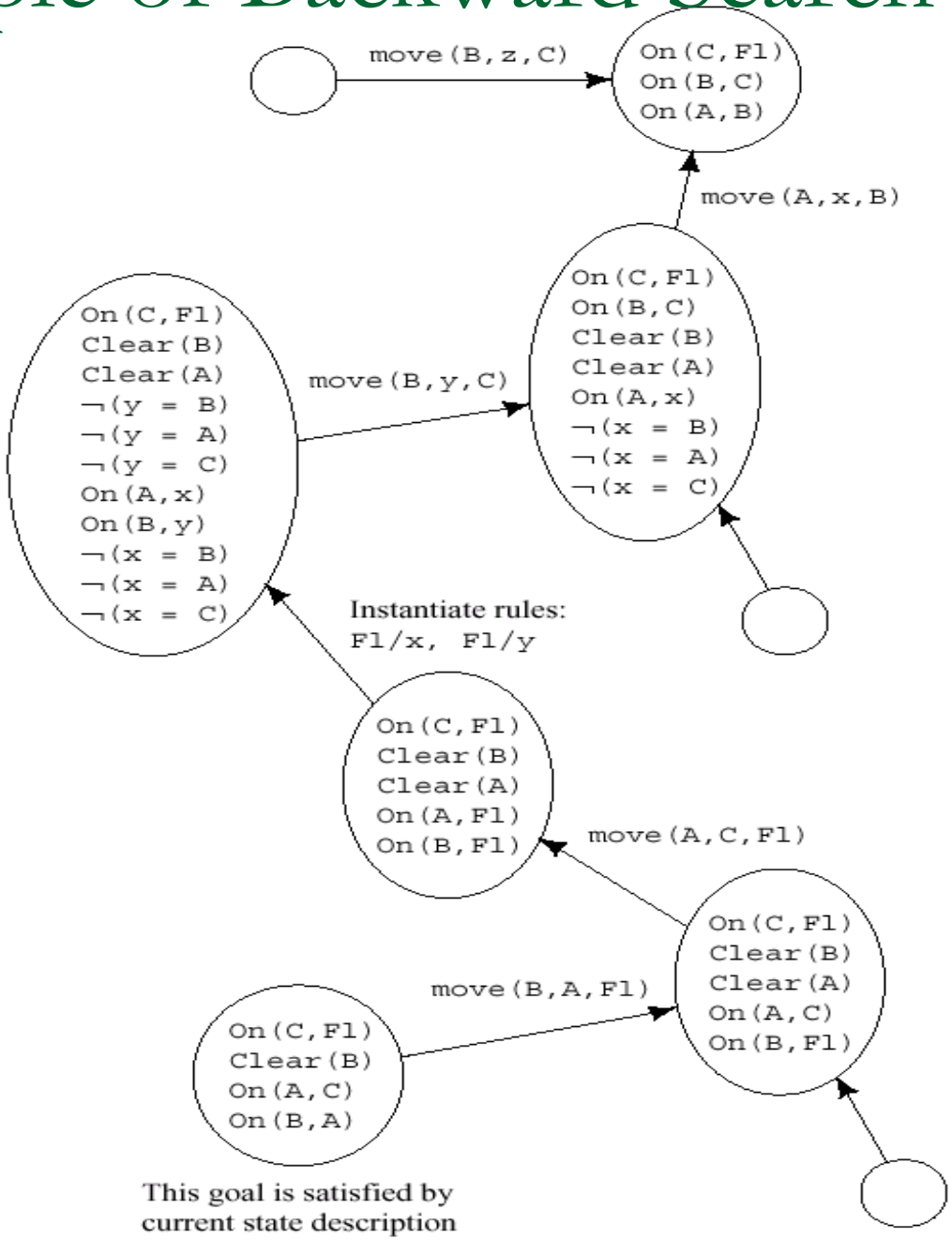




Continue until a subgoal is produced that is satisfied by current world state

# Regressing an ungrounded operator



# Example of Backward Search



# A partial order plan for putting shoes and socks

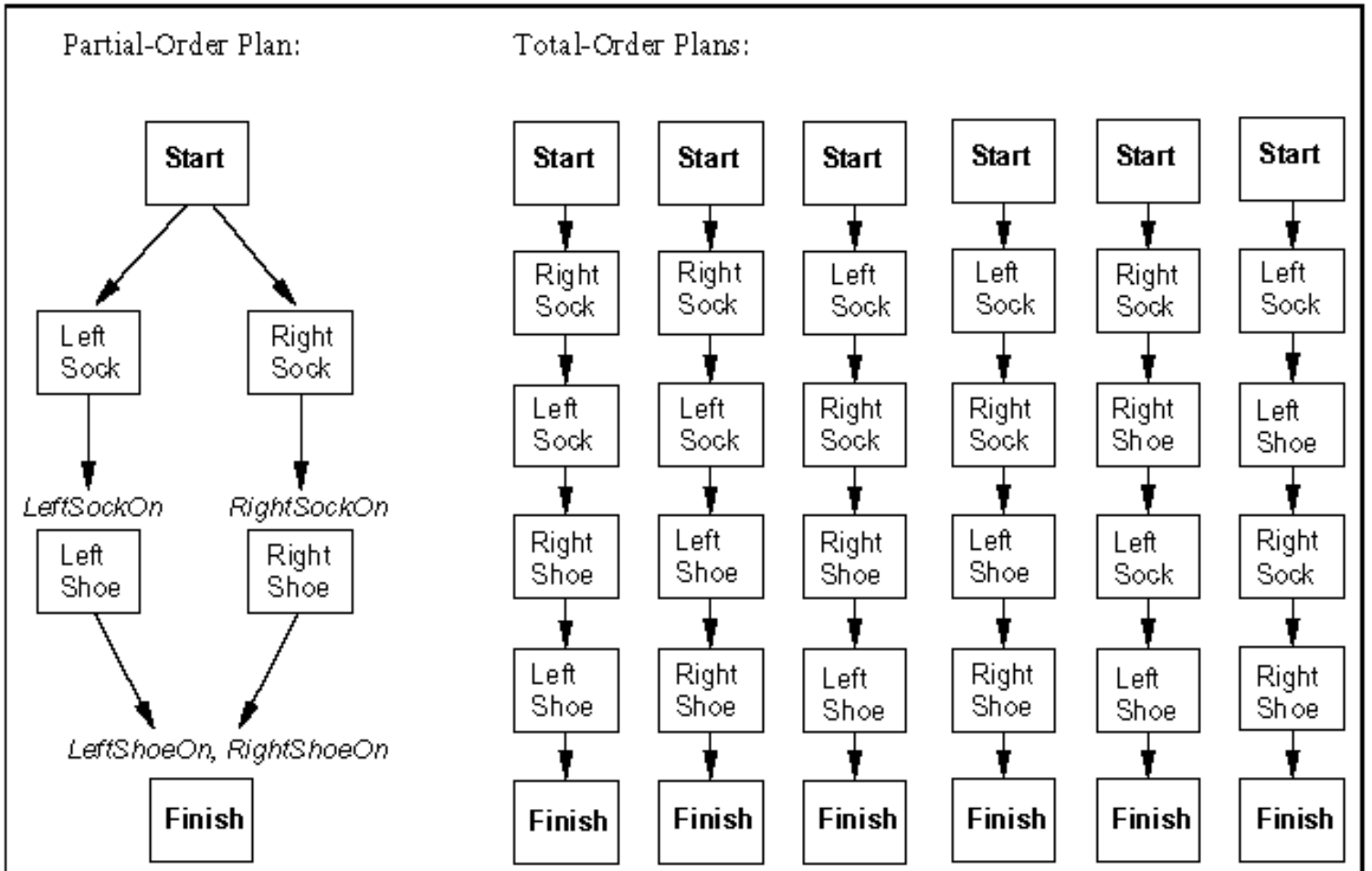



**Figure 11.6** A partial-order plan for putting on shoes and socks, and the six corresponding linearizations into total-order plans.

# Reading and Suggested Exercises

- Chapters 10
- Exercises: 10.3, 10.4, 10.15