ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Artificial Intelligence

Lecturer 12 - Planning

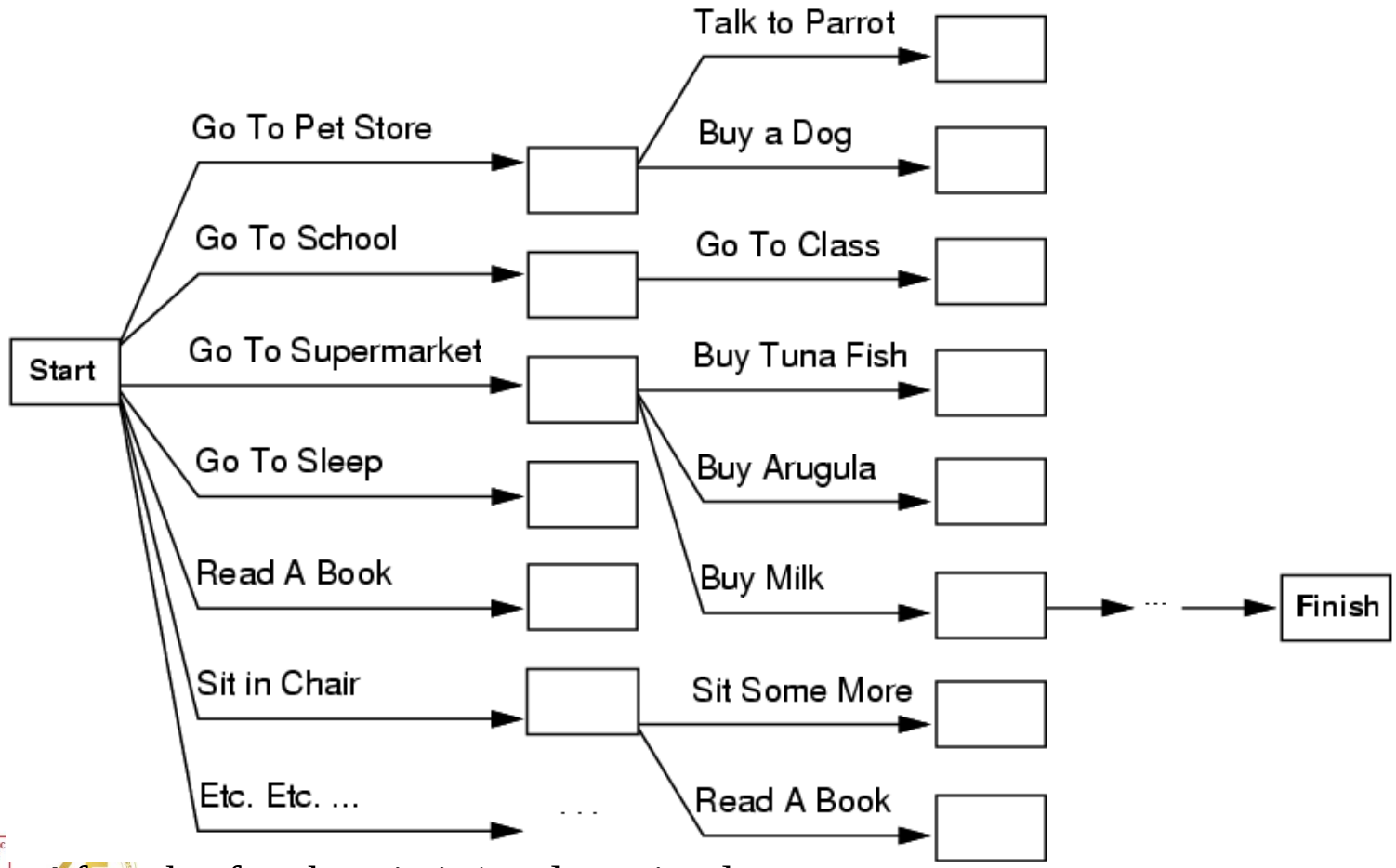
School of Information and Communication Technology - HUST

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

Problem Representation

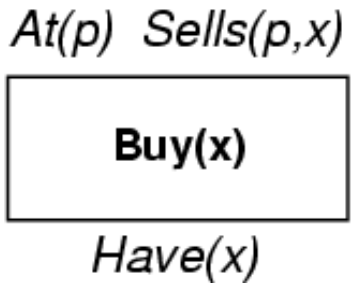
- State
  - What is true about the (hypothesized) world?
- Goal
  - What must be true in the final state of the world?
- Actions
  - What can be done to change the world?
  - Preconditions and effects

- We'll represent all these as logical predicates

STRIPS operators

- Tidily arranged actions descriptions, restricted language

- Action: *Buy*(*x*)
  - Precondition: *At*(*p*); *Sells*(*p*; *x*)
  - Effect: *Have*(*x*)



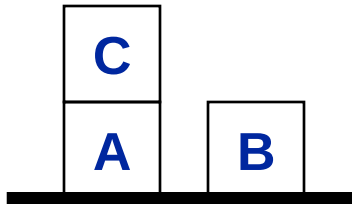
- [Note: this abstracts away many important details!]

- Restricted language $\Rightarrow$ efficient algorithm
  - Precondition: conjunction of positive literals
  - Effect: conjunction of literals
- A complete set of STRIPS operators can be translated into a set of successor-state axioms

Example: blocks world

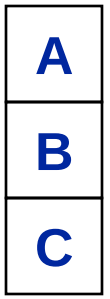
- On(b, x): block b is on x, x is another block or Table.
- MoveToTable(b, x): move block b from the top of x to Table
- Move(b, x, y): move block b from the top of x to the top of y
- Clear(x): nothing is on x

Initial



- Action (Move (b, x, y) ,
  - PRECOND: On(b, x) ∧ Clear(b) ∧ Clear(y),
  - EFFECT: On(b, y) ∧ Clear(x) ∧ ¬On(b, x) ∧ ¬Clear(y)) .
- Action(MoveToTable(b, x) ,
  - PRECOND: On(b, x) ∧ Clear(b),
  - EFFECT: On(b, Table) ∧ Clear(x) ∧ ¬On(b, x)) .

Goal

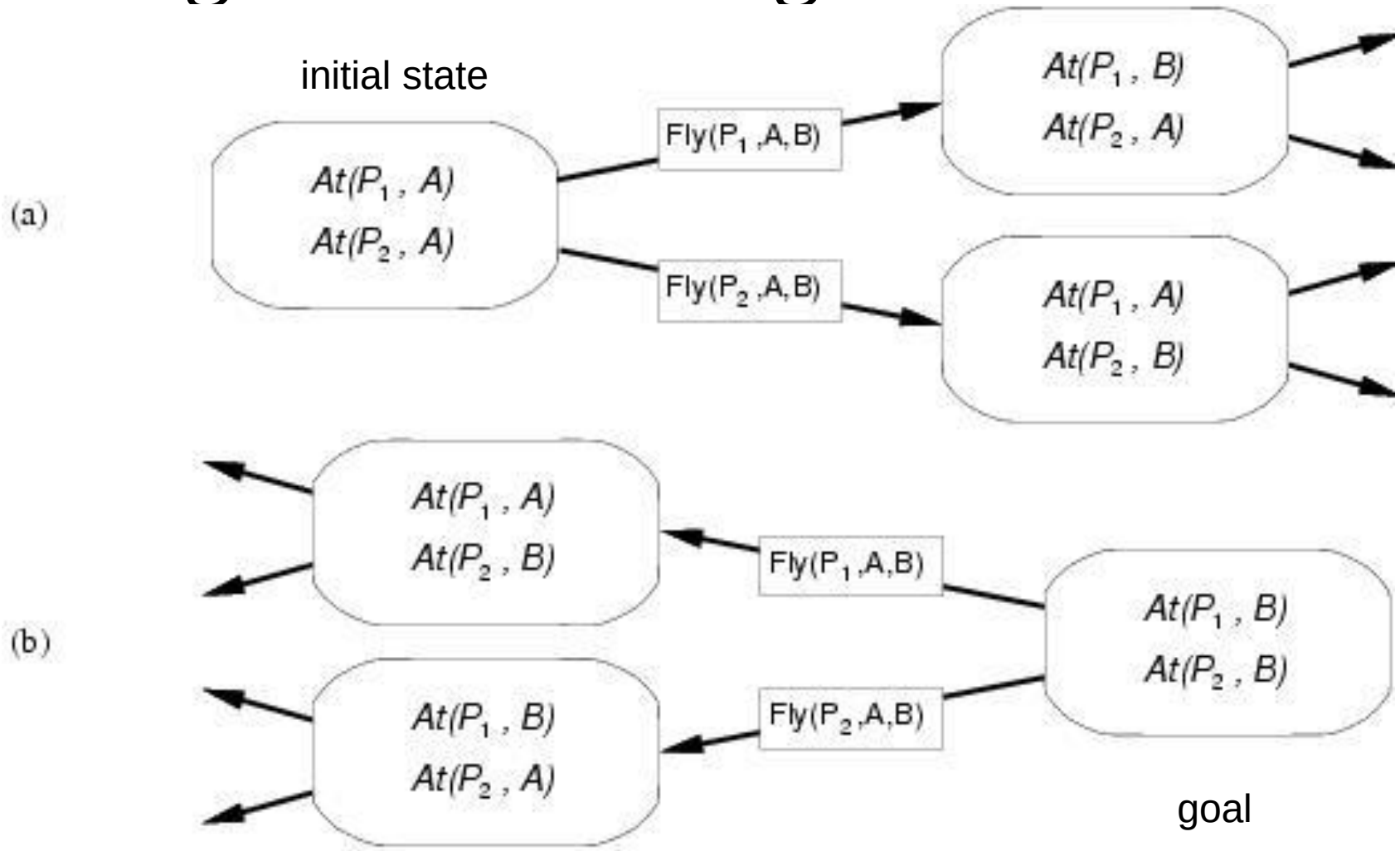


- Initial state: On(A,Table), On(C,A), On(B,Table), Clear(B), Clear(C)
- Goal: On(A,B), On(B,C)

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

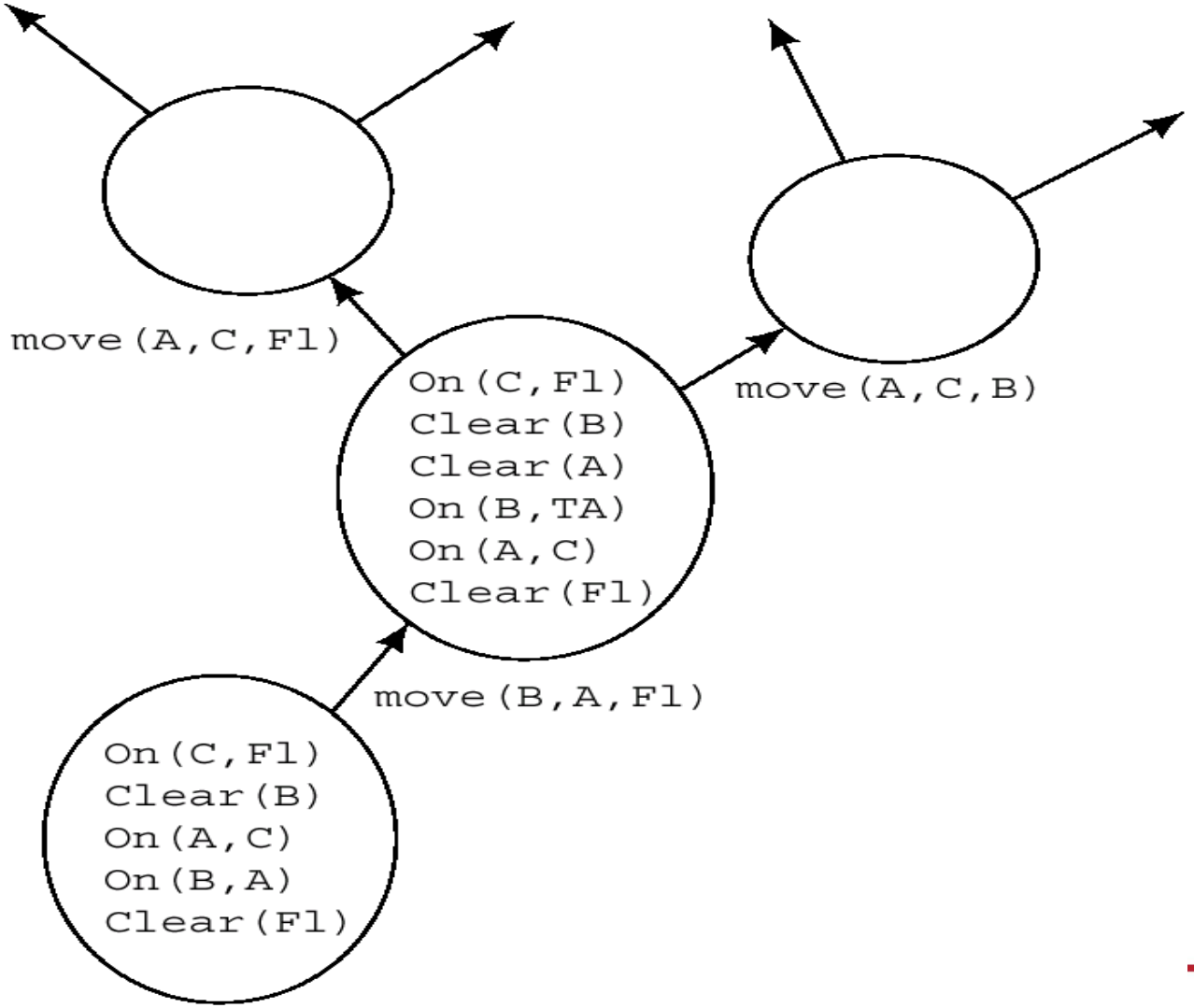
Progression and regression



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

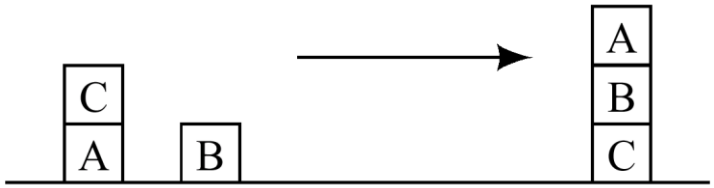
    Subgoal At(C1,B) should not be present in this state.
- Actions must not undo desired literals (consistent)
- Main advantage: only relevant actions are considered.
  - Often much lower branching factor than forward search.

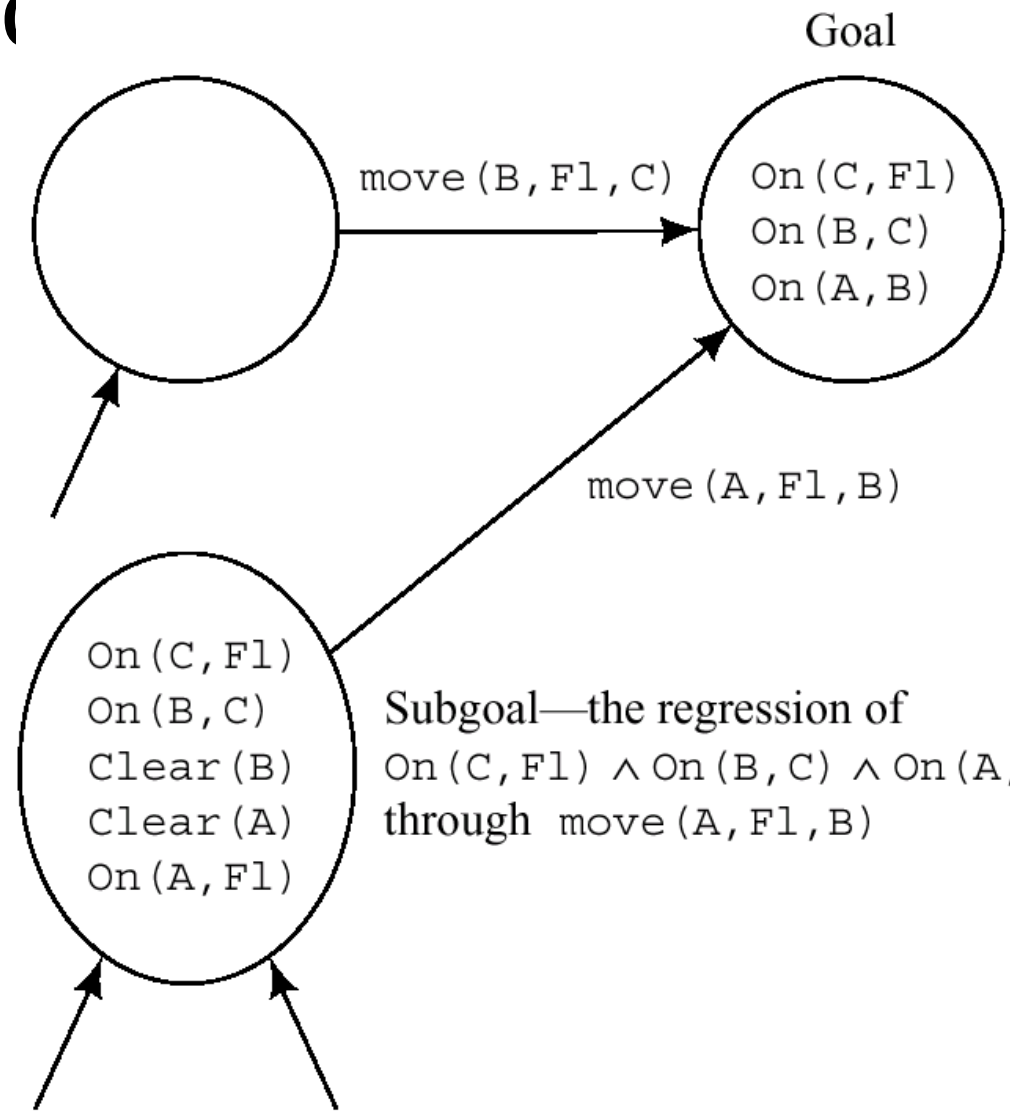
Regression algorithm

- 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

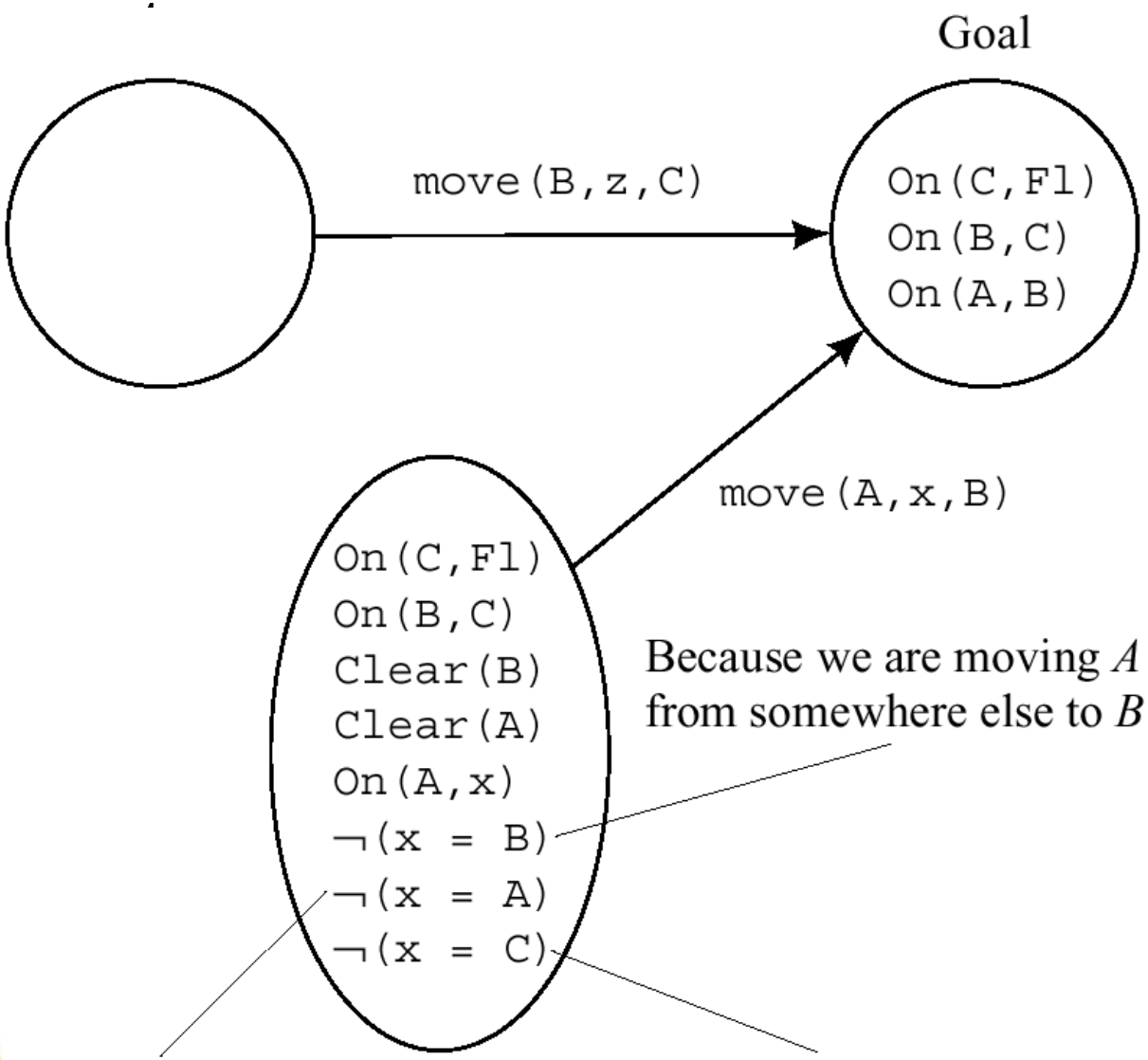
Regressing a ground operator



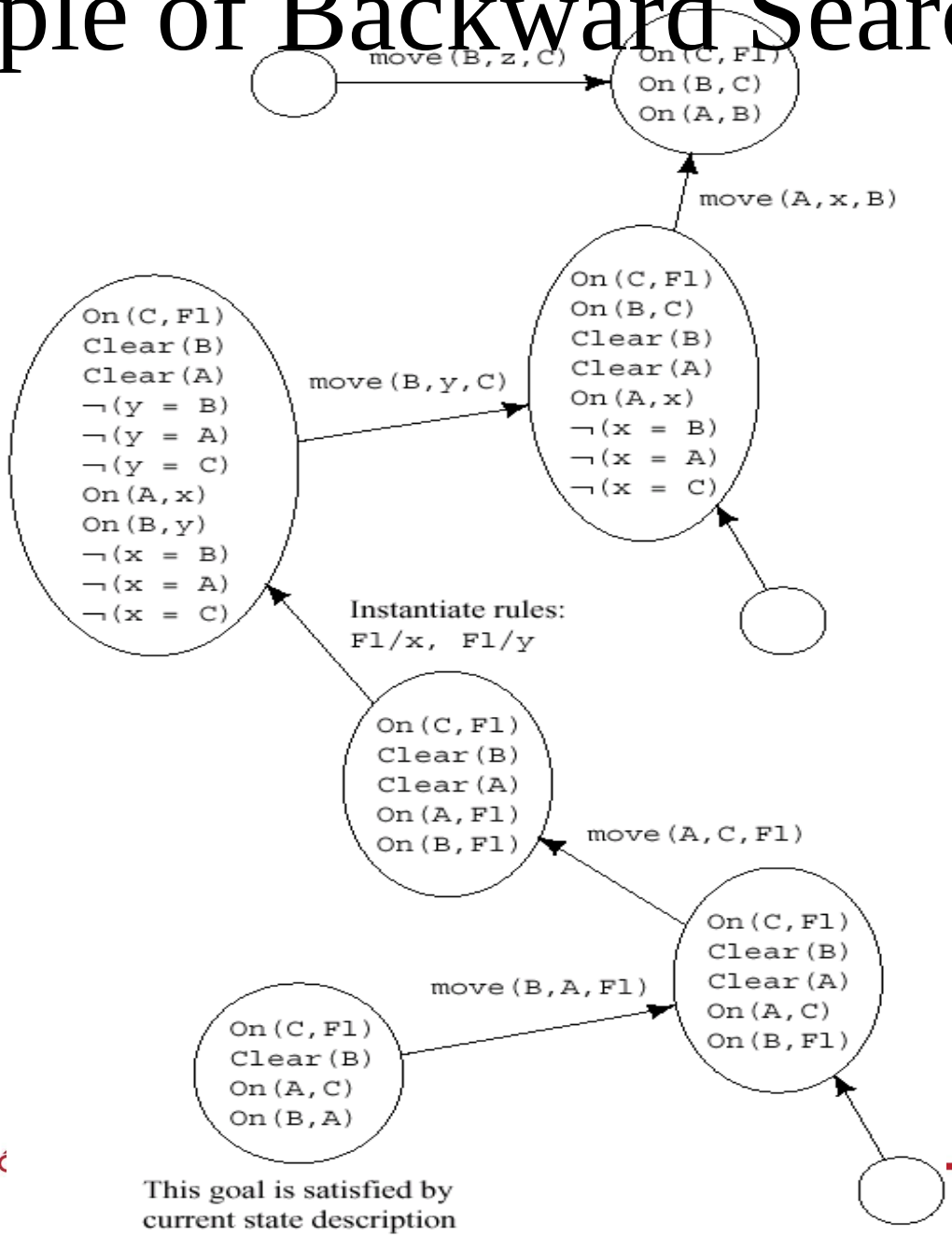


Continue until a subgoal is produced that is satisfied by current world state

Regressing an ungrounded op



Example of Backward Search



Heuristics for state-space search

- Use relax problem idea to get lower bounds on least number of actions to the goal.
  - Remove all or some preconditions
- Subgoal independence: the cost of solving a set of subgoals equal the sum cost of solving each one independently.
  - Can be pesimistic (interacting subplans)
  - Can be optimistic (negative effects)
- Simple: number of unsatisfied subgoals.
- Various ideas related to removing negative effects or positive effects.

Partial order planning

- Least commitment planning
- Nonlinear planning
- Search in the space of partial plans
- A state is a partial incomplete partially ordered plan
- Operators transform plans to other plans by:
  - Adding steps
  - Reordering
  - Grounding variables
- SNLP: Systematic Nonlinear Planning (McAllester and Rosenblitt 1991)
- NONLIN (Tate 1977)

A partial order plan for putting shoes and socks

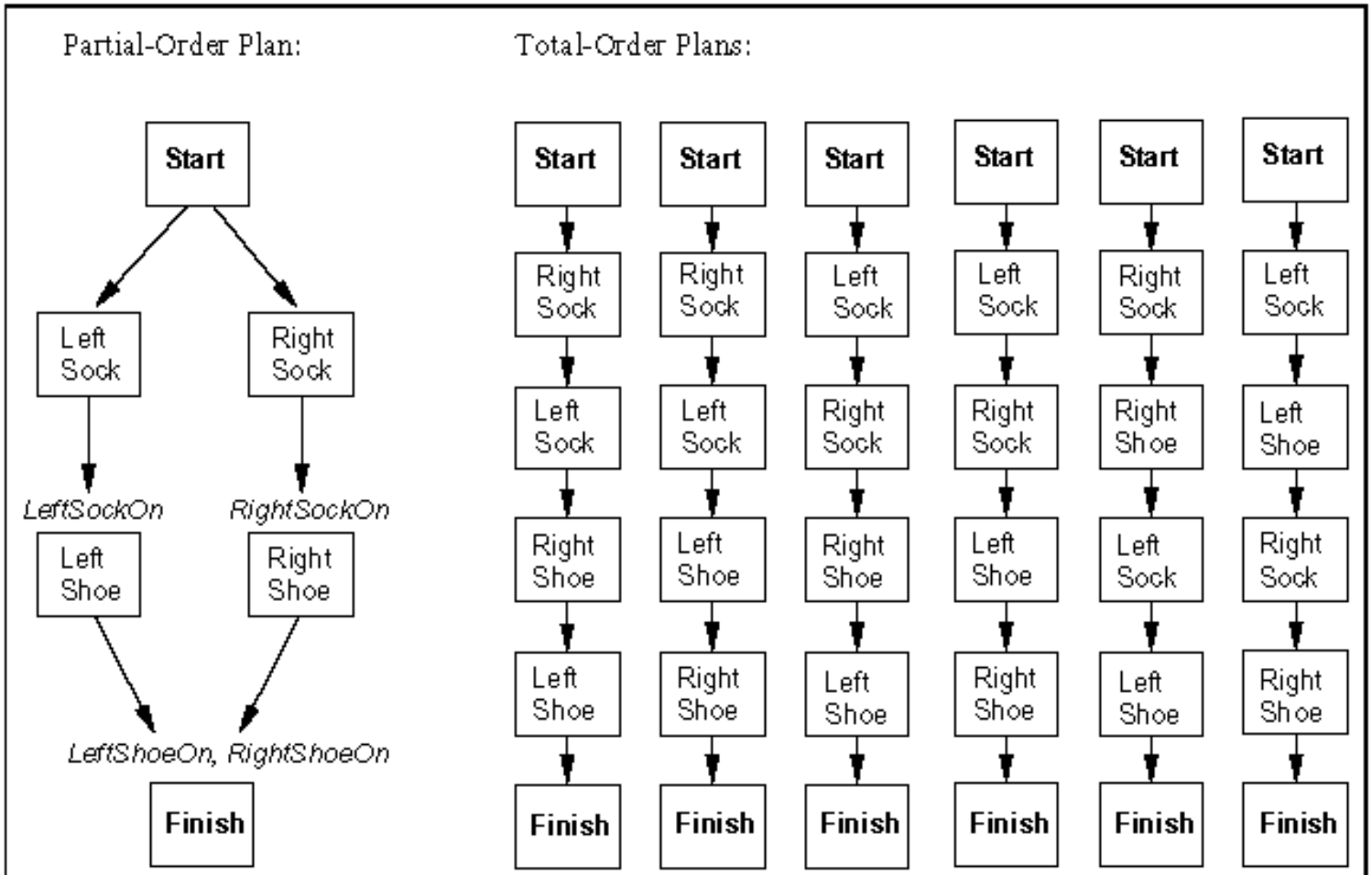
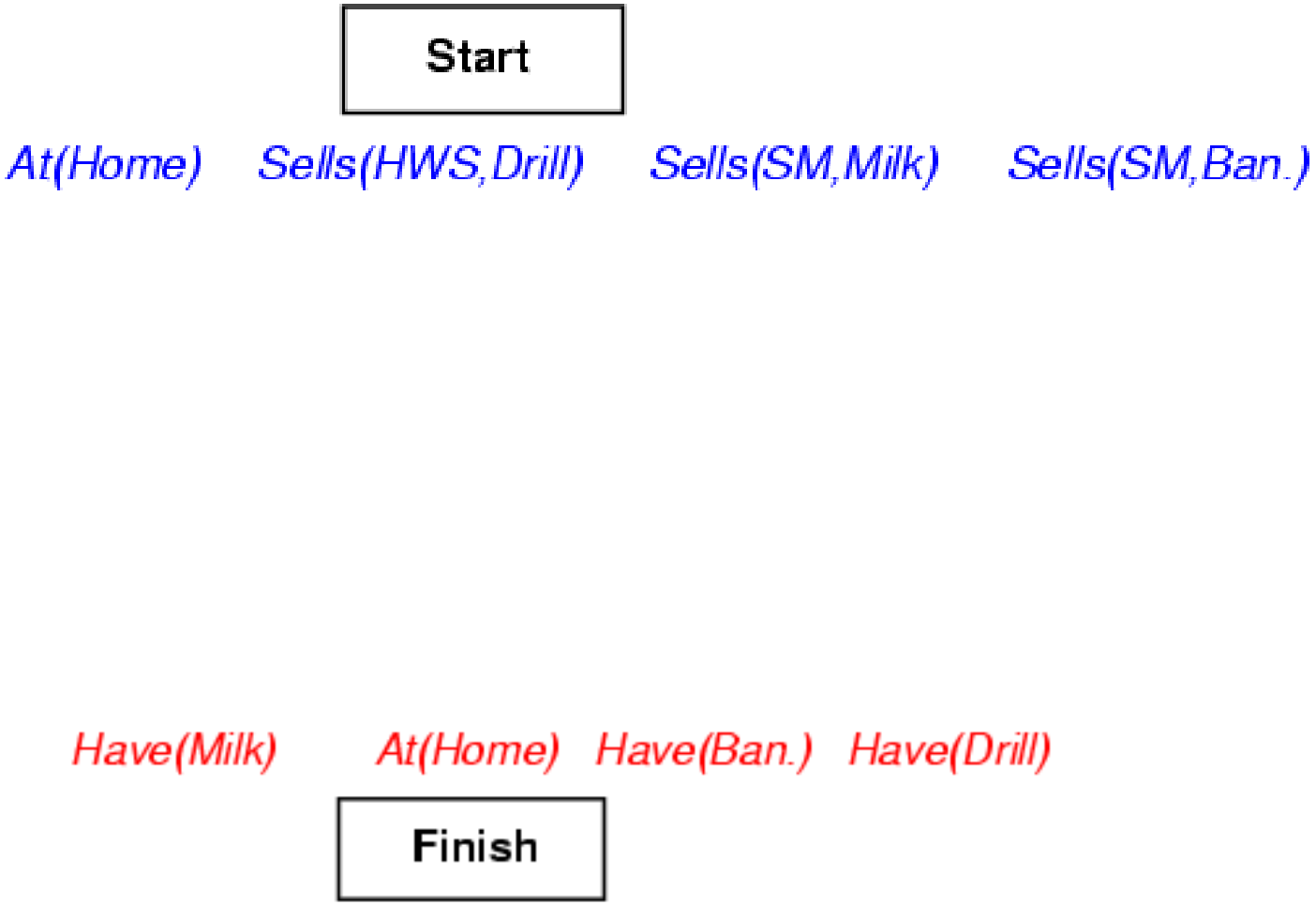


Figure 11.6 A partial-order plan for putting on shoes and socks, and the six corresponding linearizations into total-order plans.

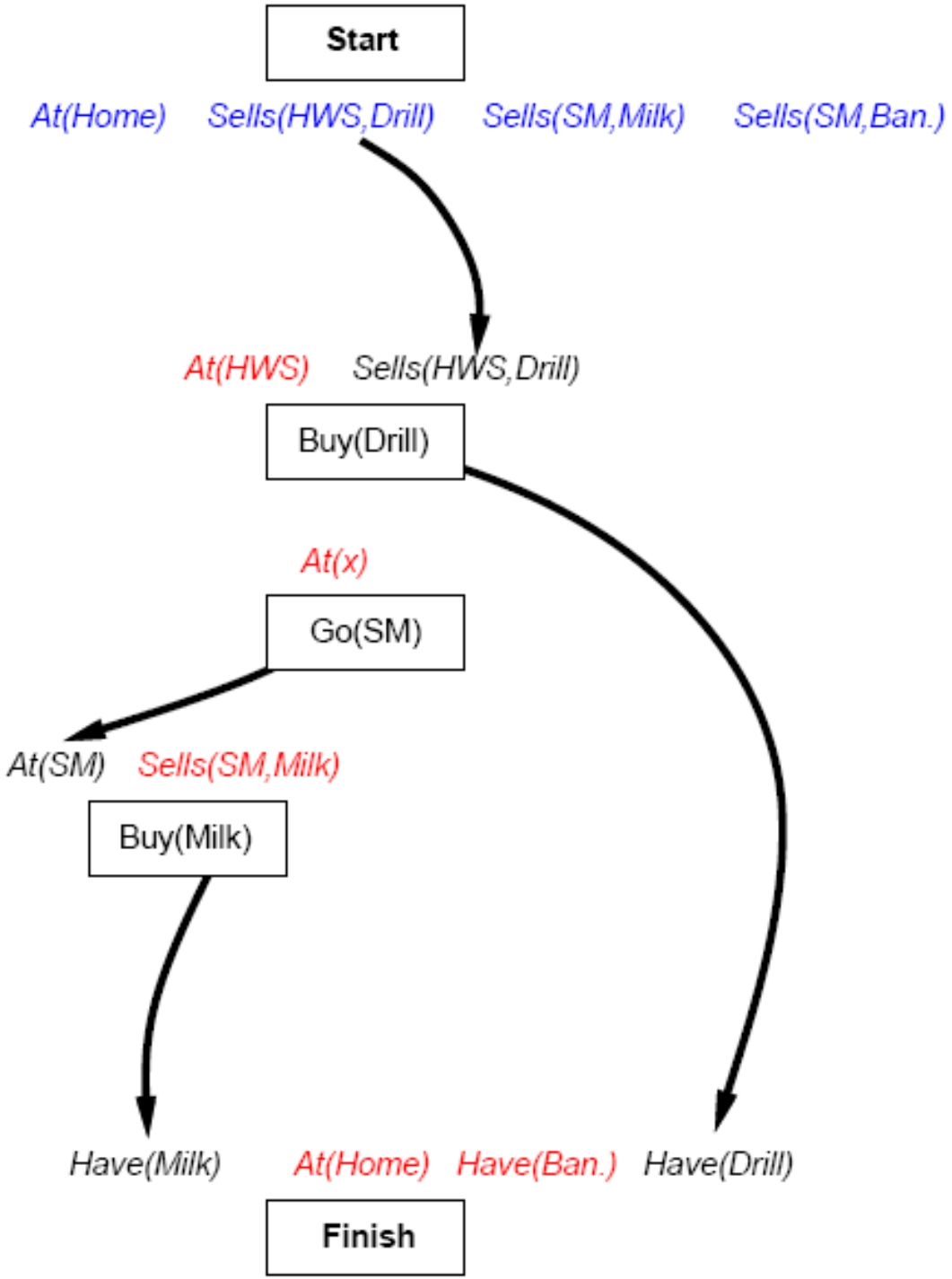
Partial-order planning

- *Partially ordered* collection of steps with
  - *Start* step has the initial state description as its effect
  - *Finish* step has the goal description as its precondition causal links from outcome of one step to precondition of another temporal ordering between pairs of steps
- Open condition = precondition of a step not yet causally linked
- A plan is complete iff every precondition is achieved
- A precondition is achieved iff it is the effect of an earlier step and no possibly intervening step undoes it

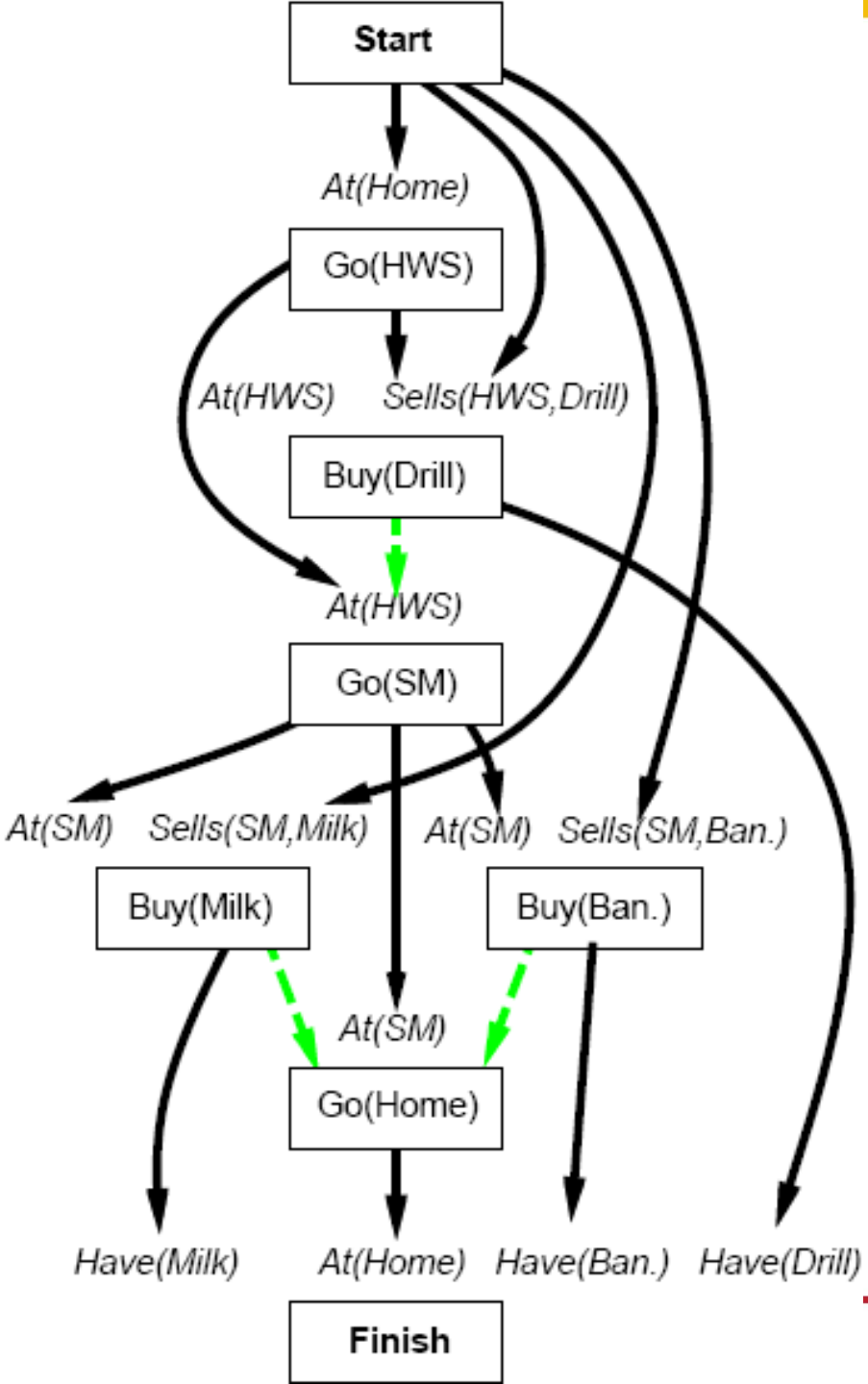
Example


Start
At(Home)
Sells(HWS,Drill)
Sells(SM,Milk)
Sells(SM,Ban.)
Have(Milk)
At(Home)
Have(Ban.)
Have(Drill)
Finish

Example


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

Example


Start
At(Home)
Go(HWS)
At(HWS)
Sells(HWS,Drill)
Buy(Drill)
At(HWS)
Go(SM)
At(SM)
Sells(SM,Milk)
At(SM)
Sells(SM,Ban.)
Buy(Milk)
Buy(Ban.)
At(SM)
Go(Home)
Have(Milk)
At(Home)
Have(Ban.)
Have(Drill)
Finish

Planning process

- Operators on partial plans:
  - add a link from an existing action to an open condition
  - add a step to fulfill an open condition
  - order one step wrt another to remove possible conflicts
- Gradually move from incomplete/vague plans to complete, correct plans
- Backtrack if an open condition is unachievable or if a conflict is unresolvable

POP algorithm sketch

function POP(*initial, goal, operators*) returns *plan*

  plan ← Make-Minimal-Plan(*initial, goal*)

  loop do

    if Solution?(*plan*) then return *plan*

    S_{need} , c ← Select-Subgoal(*plan*)

    Choose-Operator(*plan, operators*,S_{need}, c)

    Resolve-Threats(*plan*)

  end

---

function Select-Subgoal(*plan*) returns S_{need} , c

  pick a plan step S_{need} from Steps(*plan*)

    with a precondition c that has not been achieved

  return S_{need} , c

POP algorithm (con't)

procedure Choose-Operator(*plan*, *operators*,S_{need}, *c*)

choose a step S_{add} from *operators* or Steps(*plan*) that has *c* as as an effect

if there is no such step then fail

add the causal link $S_{add} \rightarrow^{c} S_{need}$ to Links(*plan*)

add the ordering constraint $S_{add} < S_{need}$ to Orderings(*plan*)

if S_{add} is a newly added step from *operators* then

add S_{add} to Steps(*plan*)

add *Start* $< S_{add} <$ *Finish* to Orderings(*plan*)

---

procedure Resolve-Threats(*plan*)

for each S_{threat} that threatens a link $S_i \rightarrow^{c} S_j$ in Links(*plan*) do

choose either

Demotion: Add $S_{threat} < S_i$ to Orderings(*plan*)

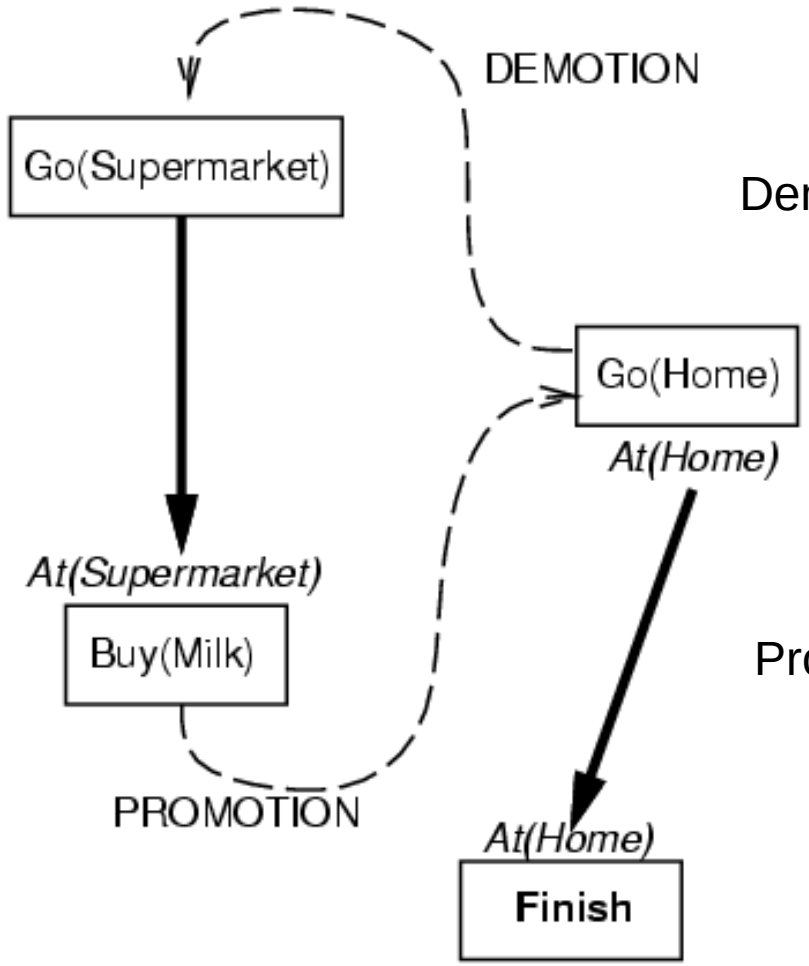
Promotion: Add $S_j < S_{threat}$ to Orderings(*plan*)

if not Consistent(*plan*) then fail

end

Clobbering and promotion/demotion

- A clobberer is a potentially intervening step that destroys the condition achieved by a causal link. E.g., *Go*(*Home*) clobbers *At*(*Supemarket):*



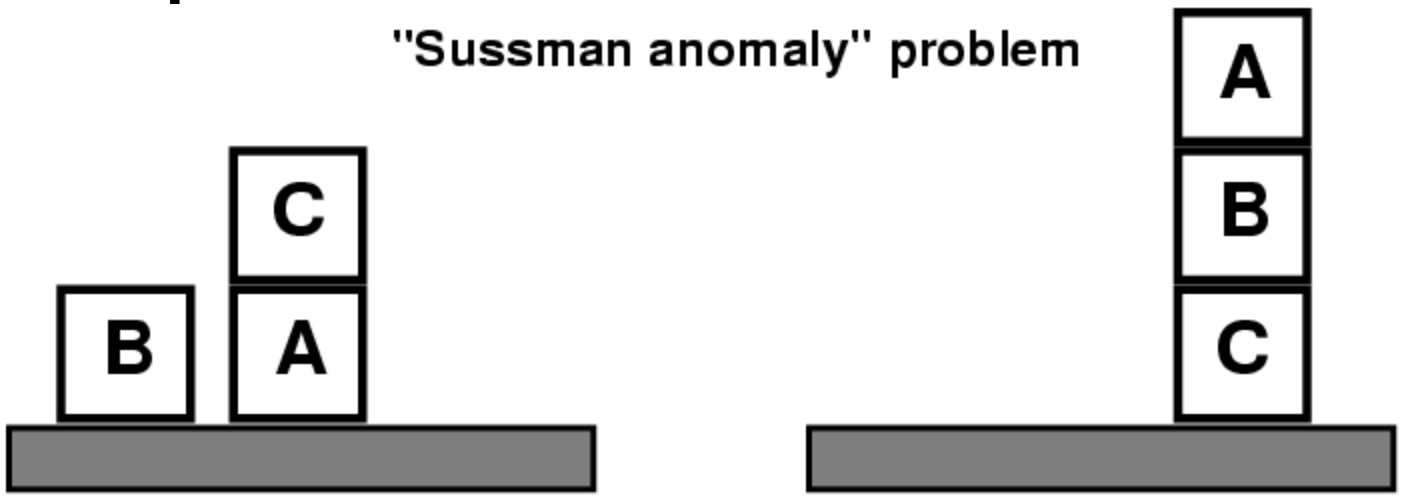
Demotion: put before *Go*(*Supermarket)*

Promotion: put after *Buy*(*Milk)*

Properties of POP

- Nondeterministic algorithm: backtracks at choice points on failure
  - choice of S_{add} to achieve S_{need}
  - choice of demotion or promotion for clobberer
  - selection of S_{need} is irrevocable
- POP is sound, complete, and systematic (no repetition)
- Extensions for disjunction, universals, negation, conditionals
- Can be made effcient with good heuristics derived from problem description
- Particularly good for problems with many loosely related subgoals

Example: Blocks world



Clear(x) On(x,z) Clear(y)

PutOn(x,y)

~On(x,z) ~Clear(y)
Clear(z) On(x,y)

Clear(x) On(x,z)

PutOnTable(x)

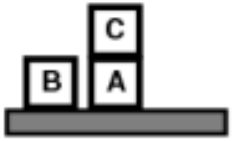
~On(x,z) Clear(z) On(x,Table)

+ several inequality constraints

Example: Blocks world

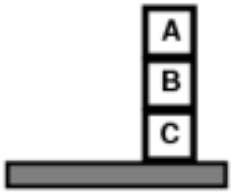
START

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

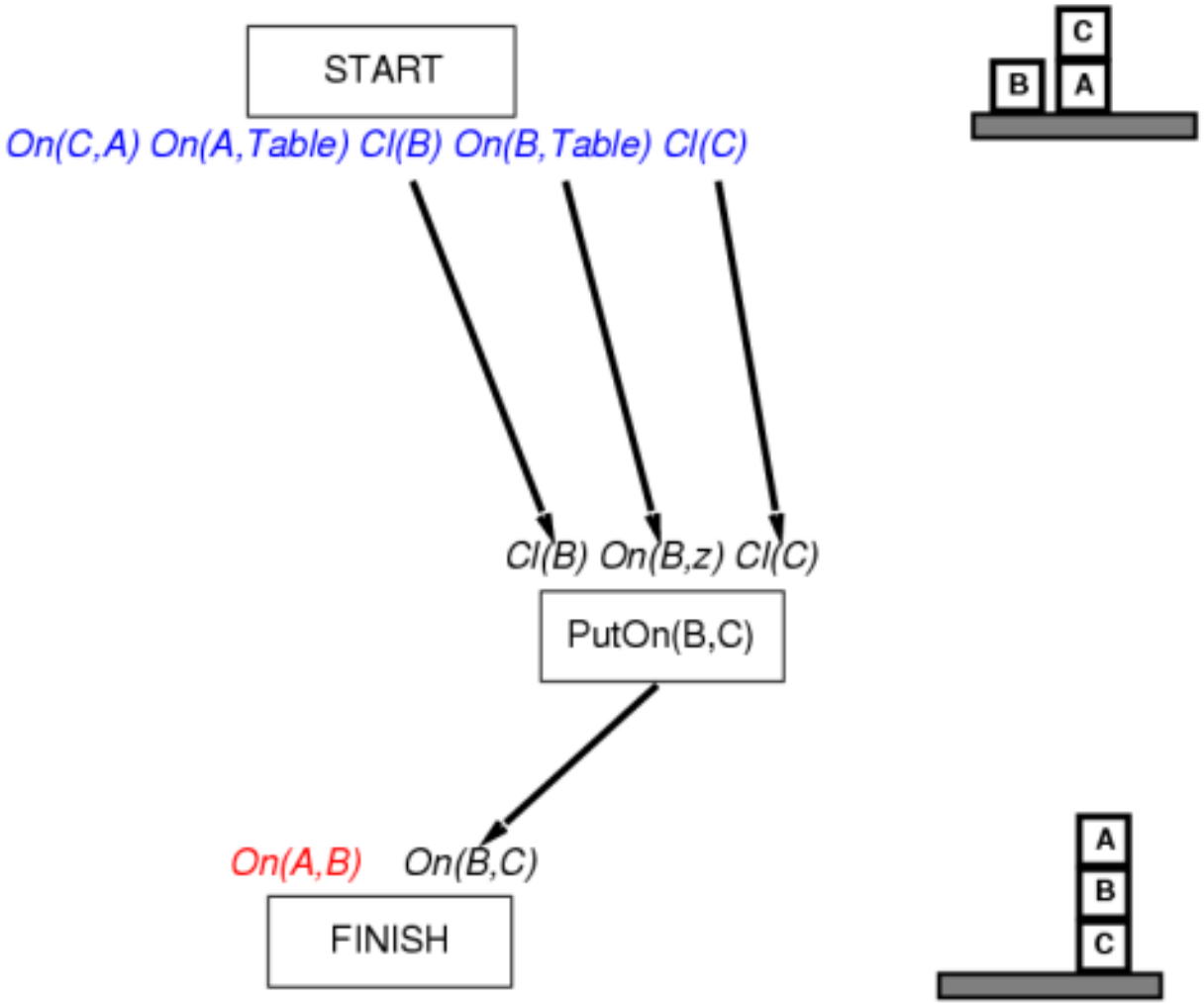


On(A,B) On(B,C)

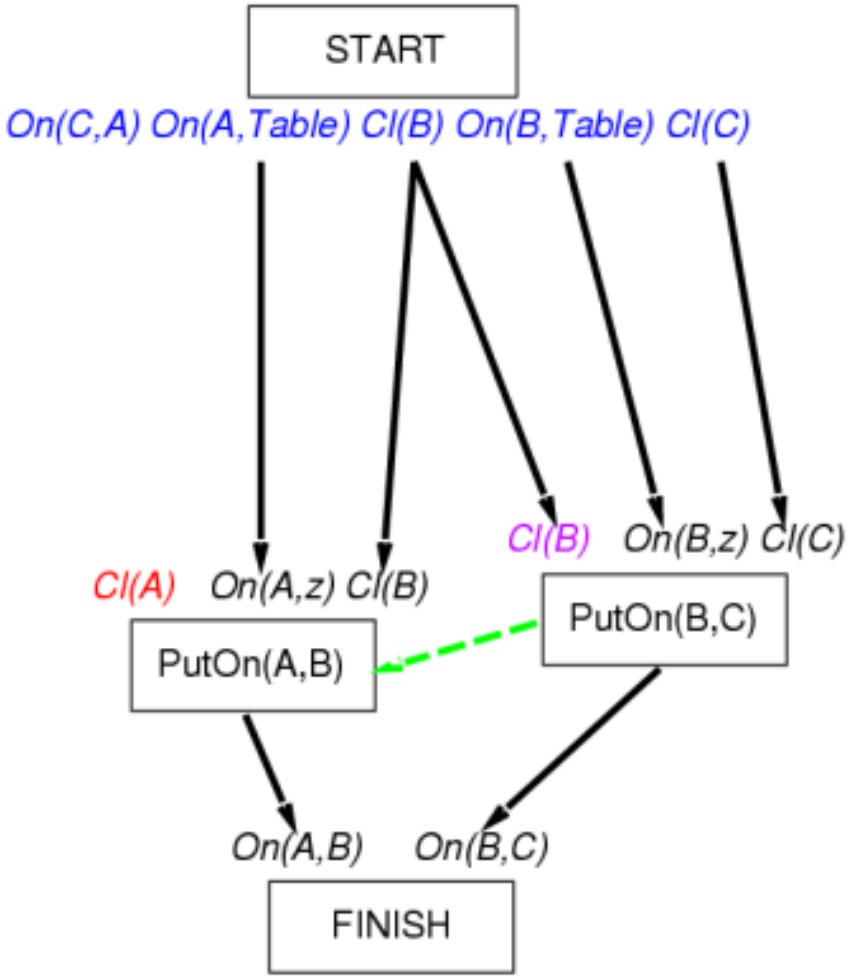
FINISH

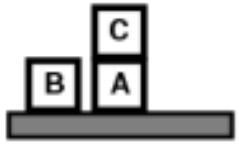


Example: Blocks world

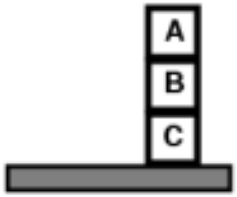


Example: Blocks world

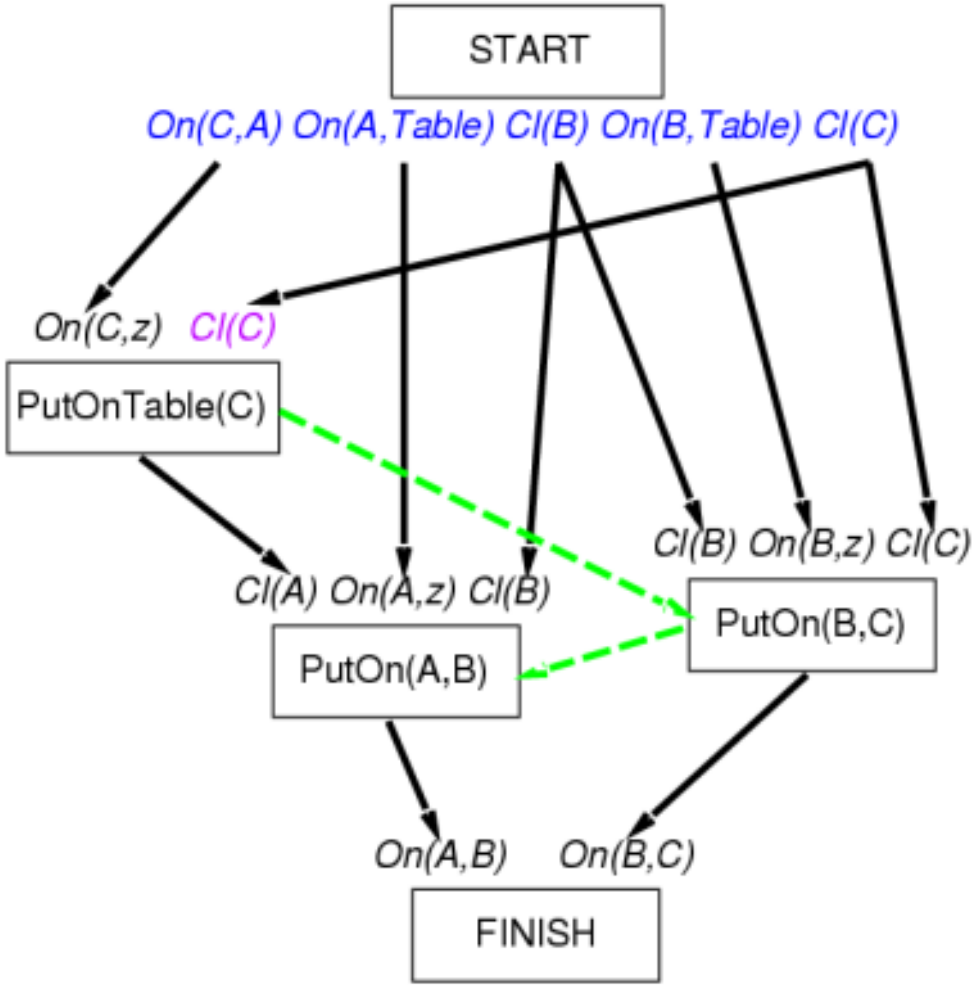


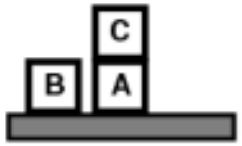


PutOn(A,B)
clobbers Cl(B)
=> order after
PutOn(B,C)



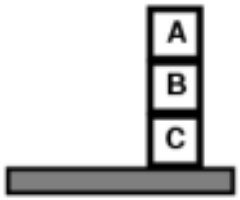
Example: Blocks world





PutOn(A,B) clobbers Cl(B) => order after PutOn(B,C)

PutOn(B,C) clobbers Cl(C) => order after PutOnTable(C)



Planning Graphs

- A planning graph consists of a sequence of levels that correspond to time-steps in the plan
- Level 0 is the initial state.
- Each level contains a set of literals and a set of actions
- Literals are those that could be true at the time step.
- Actions are those that their preconditions could be satisfied at the time step.
- Works only for propositional planning.

Example:Have cake and eat it too

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg Have(Cake)$
EFFECT: $Have(Cake)$

Figure 11.11 The "have cake and eat cake too" problem.

The Planning graphs for “have cake”,

- Persistence actions: Represent “inactions” by boxes: frame axiom
- Mutual exclusions (mutex) are represented between literals and actions.
- S1 represents multiple states
- Continue until two levels are identical. The graph levels off.
- The graph records the impossibility of certain choices using mutex links.
- Complexity of graph generation: polynomial in number of literals.

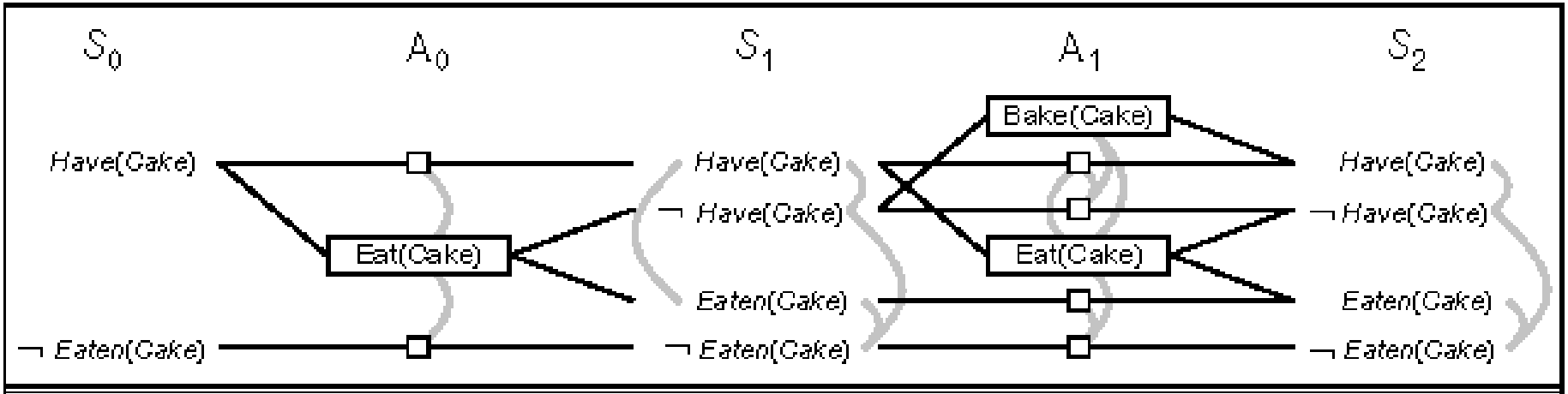


Figure 11.12 The planning graph for the “have cake and eat cake too” problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions) and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines.

Defining Mutex relations

- A mutex relation holds between two actions on the same level iff any of the following holds:
  - **Inconsistency effect:one** action negates the effect of another. Example “eat cake and presistence of have cake”
  - **Interference:** One of the effect of one action is the negation of the precondition of the other. Example: eat cake and persistence of Have cake
  - **Competing needs**: one of the preconditions of one action is mutually exclusive with a precondition of another. Example: Bake(cake) and Eat(Cake).
  - A mutex relation holds between 2 literals at the same level iff one is the negation of the other or if each possible pair of actions that can achieve the 2 literals is mutually exclusive.

Planning graphs for heuristic estimation

- Estimate the cost of achieving a goal by the level in the planning graph where it appears.
- To estimate the cost of a conjunction of goals use one of the following:
  - Max-level: take the maximum level of any goal (admissible)
  - Sum-cost: Take the sum of levels (inadmissible)
  - Set-level: find the level where they all appear without Mutex
- Graph plans are relaxation of the problem.
- Representing more than pair-wise mutex is not cost-effective

The graphplan algorithm

```
function GRAPHPLAN(problem) returns solution or failure

    graph ← INITIAL-PLANNING-GRAPH(problem)
    goals ← GOALS[problem]
    loop do
        if goals all non-mutex in last level of graph then do
            solution ← EXTRACT-SOLUTION(graph, goals, LENGTH(graph))
            if solution ≠ failure then return solution
            else if NO-SOLUTION-POSSIBLE(graph) then return failure
        graph ← EXPAND-GRAPH(graph, problem)
```

Figure 11.13 The GRAPHPLAN algorithm. GRAPHPLAN alternates between a solution extraction step and a graph expansion step. EXTRACT-SOLUTION looks for whether a plan can be found, starting at the end and searching backwards. EXPAND-GRAPH adds the actions for the current level and the state literals for the next level.

Planning graph for spare tire a S2
goal: at(spare,axle)

- S2 has all goals and no mutex so we can try to extract solutions
- Use either CSP algorithm with actions as variables
- Or search backwards

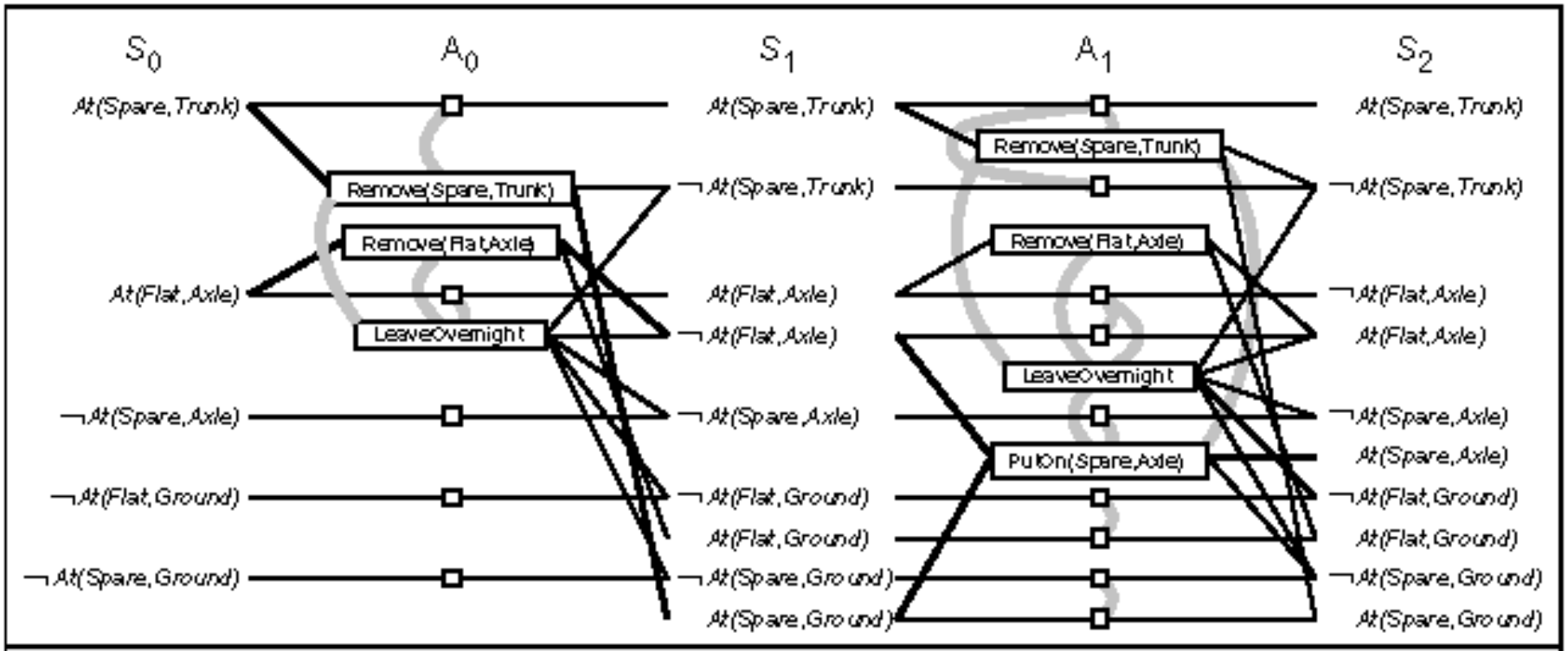


Figure 11.14 The planning graph for the spare tire problem after expansion to level S_2. Mutex links are shown as gray lines. Only some representative mutexes are shown, because the graph would be too cluttered if we showed them all. The solution is indicated by bold lines and outlines.

Search planning-graph backwards with heuristics

- How to choose an action during backwards search:
- Use greedy algorithm based on the level cost of the literals.
- For any set of goals:
- 1. Pick first the literal with the highest level cost.
- 2. To achieve the literal, choose the action with the easiest preconditions first (based on sum or max level of precond literals).

Properties of planning graphs; termination

- Literals increase monotonically
  - Once a literal is in a level it will persist to the next level
- Actions increase monotonically
  - Since the precondition of an action was satisfied at a level and literals persist the action's precond will be satisfied from now on
- Mutexes decrease monotonically:
  - If two actions are mutex at level Si, they will be mutex at all previous levels at which they both appear
- Because literals increase and mutex decrease it is guaranteed that we will have a level where all goals are non-mutex

Planning with propositional logic

- Express propositional planning as a set of propositions.
- Index propositions with time steps:
- On(A,B)_0, ON(B,C)_0
- Goal conditions: the goal conjuncts at time T, T is determined arbitrarily.
- Unknown propositions are not stated.
- Propositions known not to be true are stated negatively.
- Actions: a proposition for each action for each time slot.
- Succesor state axioms need to be expressed for each action (like in the situation calculus but it is propositional)

Planning with propositional logic (continued)

- We write the formula:
  - Initial state and successor state axioms and goal
- We search for a model to the formula. Those actions that are assigned true constitute a plan.
- To have a single plan we may have a mutual exclusion for all actions in the same time slot.
- We can also choose to allow partial order plans and only write exclusions between actions that interfere with each other.
- Planning: iteratively try to find longer and longer plans.

SATplan algorithm

```
function SATPLAN(problem, T_max) returns solution or failure
  inputs: problem, a planning problem
          T_max, an upper limit for plan length

  for T = 0 to T_max do
    cnf, mapping ← TRANSLATE-TO-SAT(problem, T)
    assignment ← SAT-SOLVER(cnf)
    if assignment is not null then
      return EXTRACT-SOLUTION(assignment, mapping)
  return failure
```

Figure 11.15 The SATPLAN algorithm. The planning problem is translated into a CNF sentence in which the goal is asserted to hold at a fixed time step T and axioms are included for each time step up to T. (Details of the translation are given in the text.) If the satisfiability algorithm finds a model, then a plan is extracted by looking at those proposition symbols that refer to actions and are assigned *true* in the model. If no model exists, then the process is repeated with the goal moved one step later.

Complexity of satplan

- The total number of action symbols is:
  - |T|x|Act|x|O|^p
  - O = number of objects, p is scope of atoms.
- Number of clauses is higher.
- Example: 10 time steps, 12 planes, 30 airports, the complete action exclusion axiom has 583 million clauses.