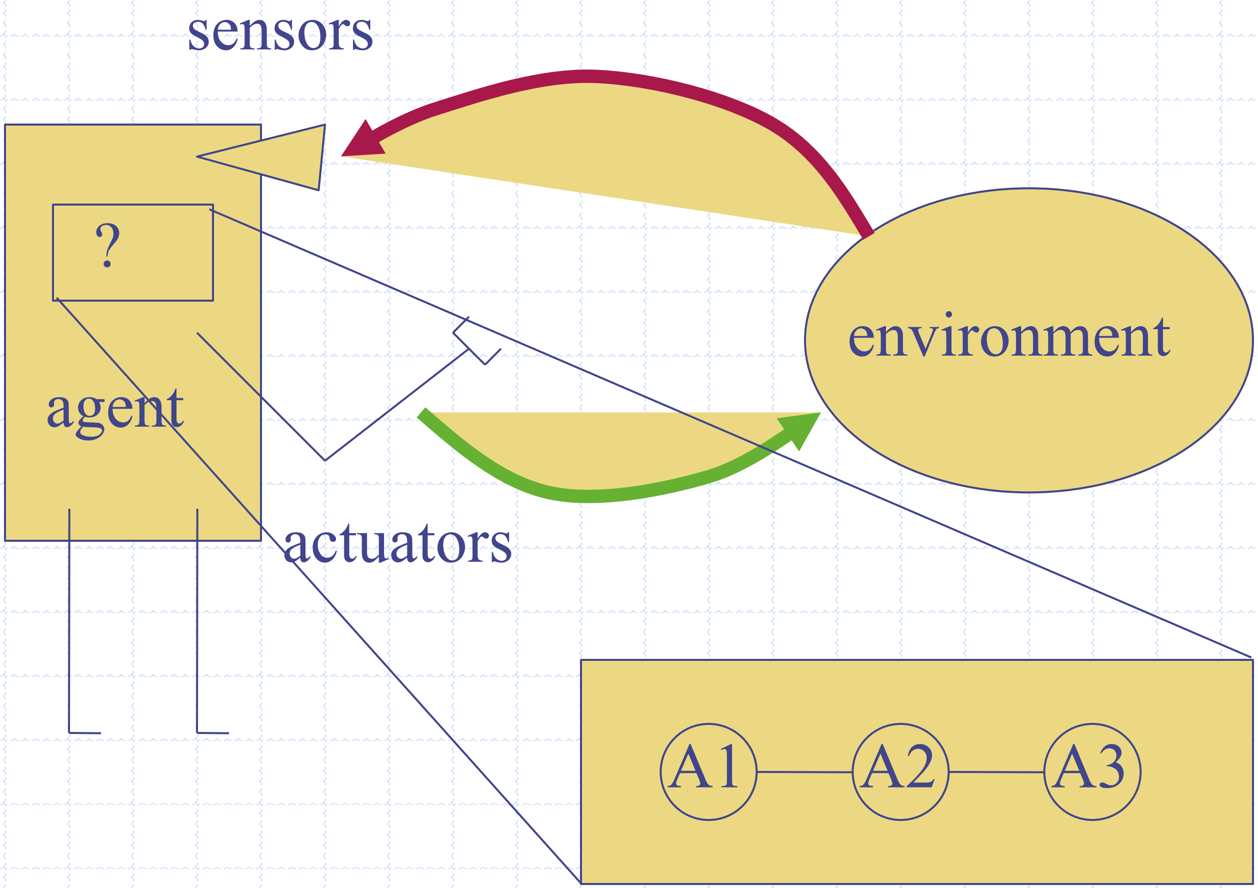
Artificial Intelligence
Planning

Planning Agent



Outline

- The Planning problem
- Planning with State-space search
- Partial-order planning
- Planning graphs

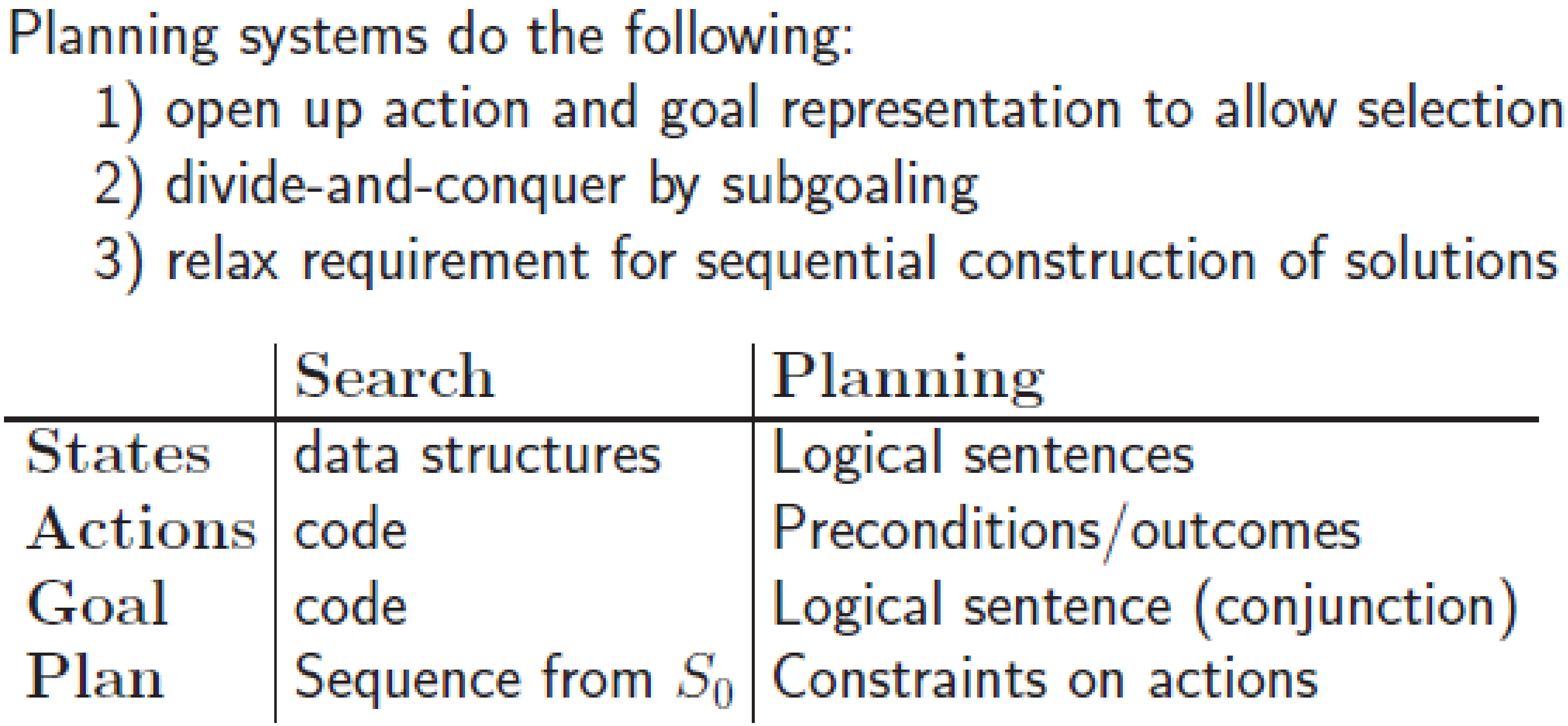
Planning problem

- Classical planning environment: fully observable, deterministic, finite, static and discrete.
- Find a sequence of actions that achieves a given goal when executed from a given initial world state. That is, given
  - a set of action descriptions (defining the possible primitive actions by the agent),
  - an initial state description, and
  - a goal state description or predicate,
- compute a plan, which is
  - a sequence of action instances, such that executing them in the initial state will change the world to a state satisfying the goal-state description.
- Goals are usually specified as a conjunction of subgoals to be achieved

Planning vs. problem solving

- Planning and problem solving methods can often solve the same sorts of problems
- Planning is more powerful because of the representations and methods used
- States, goals, and actions are decomposed into sets of sentences (usually in first-order logic)
- Search often proceeds through plan space rather than state space (though first we will talk about state-space planners)
- Subgoals can be planned independently, reducing the complexity of the planning problem

Planning vs. problem solving



Planning systems do the following:

1) open up action and goal representation to allow selection
2) divide-and-conquer by subgoaling
3) relax requirement for sequential construction of solutions

	Search	**Planning**
States	data structures	Logical sentences
Actions	code	Preconditions/outcomes
Goal	code	Logical sentence (conjunction)
Plan	Sequence from S_0	Constraints on actions

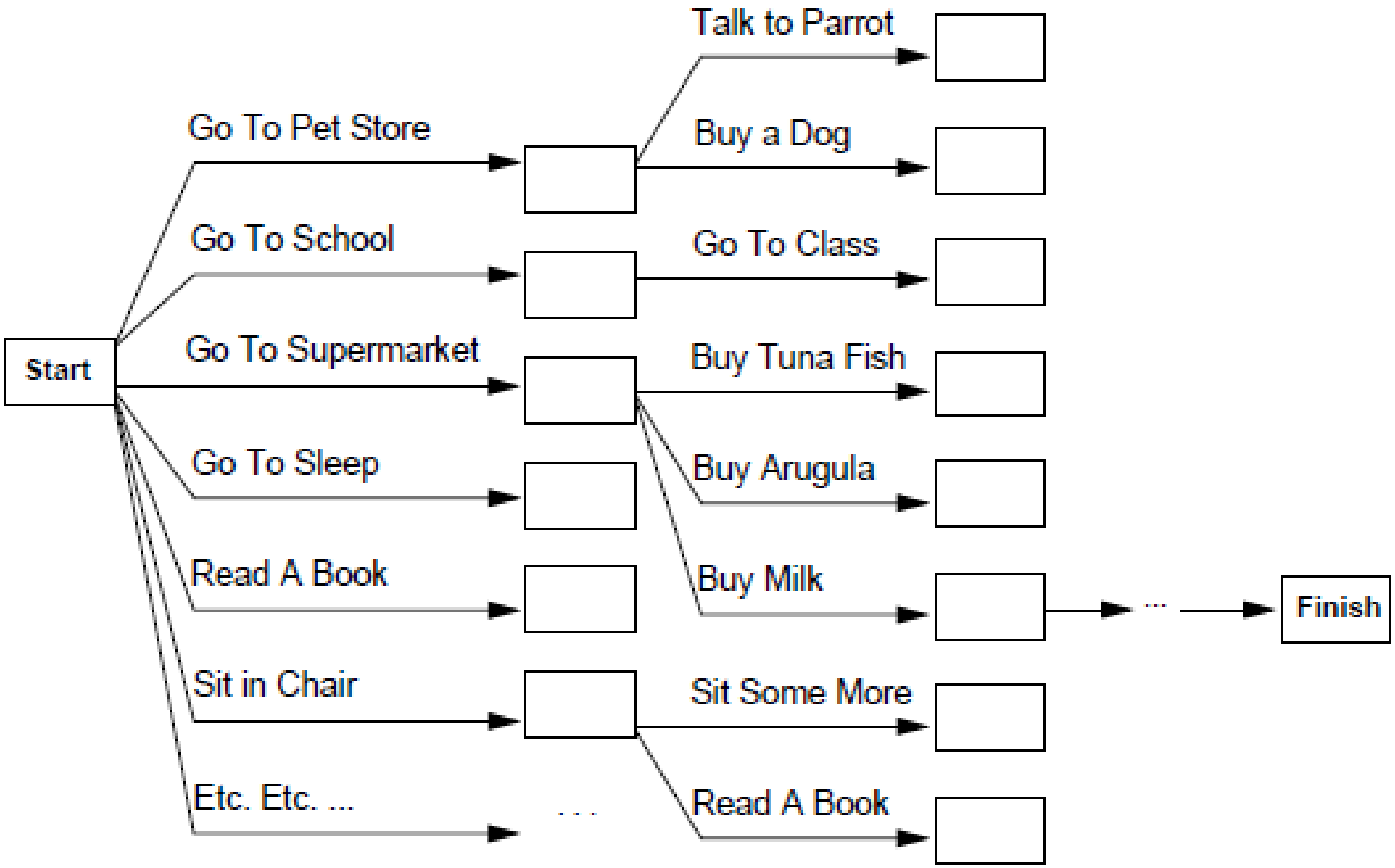
Goal of Planning

- Choose actions to achieve a certain goal
- But isn’t it exactly the same goal as for problem solving?
- Some difficulties with problem solving:
  - The successor function is a black box: it must be “applied” to a state to know which actions are possible in that state and what are the effects of each one

Have(Milk)

- Suppose that the goal is HAVE(MILK).
  - From some initial state where HAVE(MILK) is not satisfied, the successor function must be repeatedly applied to eventually generate a state where HAVE(MILK) is satisfied.
  - An explicit representation of the possible actions and their effects would help the problem solver select the relevant actions
  - Otherwise, in the real world an agent would be overwhelmed by irrelevant actions

Have(Milk)


Start
Go To Pet Store
Go To School
Go To Supermarket
Go To Sleep
Read A Book
Sit in Chair
Etc. Etc. ...
. . .
Talk to Parrot
Buy a Dog
Go To Class
Buy Tuna Fish
Buy Arugula
Buy Milk
Sit Some More
Read A Book
...
Finish

Planning vs Problem Solving

- Another difficulty with problem solving:
  - The goal test is another black-box function, states are domain-specific data structures, and heuristics must be supplied for each new problem
  - Suppose that the goal is HAVE(MILK)∧HAVE(BOOK)
  - Without an explicit representation of the goal, the problem solver cannot know that a state where HAVE(MILK) is already achieved is more promising than a state where neither HAVE(MILK) nor HAVE(BOOK) is achieved

Planning vs Problem Solving

- A third difficulty with problem solving:
  - The goal may consist of several nearly independent subgoals, but there is no way for the problem solver to know it
  - HAVE(MILK) and HAVE(BOOK) may be achieved by
    two nearly independent sequences of actions

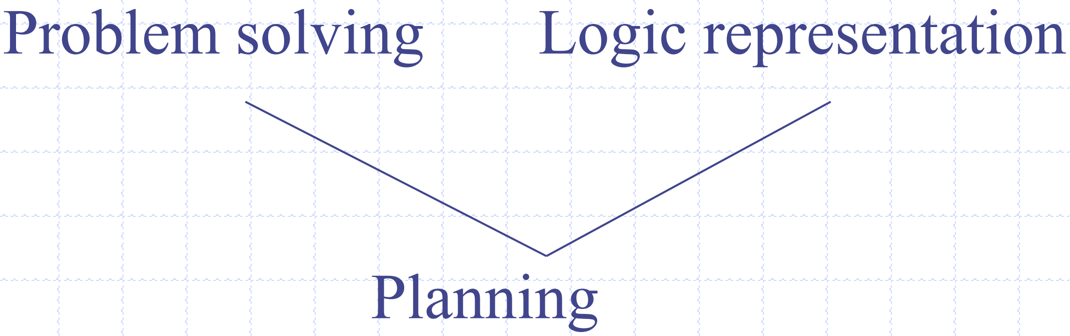
Applications of Planning

- Military operations
- Construction tasks
- Machining tasks
- M
- D
- C

Most applied systems use extended representation languages, nonlinear planning techniques, and domain-specific heuristics

Representations in Planning

- Planning opens up the black-boxes by using logic to represent:
  - Actions
  - States
  - Goals



Problem solving

Logic representation

Planning

Planning language

- What is a good language?
- Must represent
  - States
  - Goals
  - Action.
- Must be
  - Expressive enough to describe a wide variety of problems.
  - Restrictive enough to allow efficient algorithms to operate.

Languages for Planning Problems

- STRIPS
  - Stanford Research Institute Problem Solver
  - Historically important
- ADL
  - Action Description Languages
  - Relaxed some of the restrictions that made STRIPS inadequate for real-world problems
- PDDL
  - Planning Domain Definition Language
  - Revised & enhanced for the needs of the International Planning Competition
  - Currently version 3.1
  - Includes STRIPS and ADL

STRIPS General language features

- Representation of states
  - Decompose world in logical conditions and represent state as conjunction of positive literals.
    - Propositional literals: Poor ∧ Unknown
    - FO-literals (grounded and function-free):
      - at(plane1, phl) ∧ at(plane2, bwi)
      - does *not* allow
        - at(X,Y), (not grounded)
        - at(father(fred),bwi), (function)
        - ¬at(plane1, phl. (negative literal)

STRIPS Cont

- Closed world assumption
- Representation of goals
  - Partially specified state and represented as a conjunction of positive ground literals
  - A goal is satisfied if the state contains all literals in goal.
  - e.g.: at(paula, phl) ∧ at(john,bwi) satisfies the goal at(paula, phl)

General language features

- Representations of actions
  - Action = PRECOND + EFFECT
    - Action(Fly(p,from, to),
    - PRECOND: At(p,from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
    - EFFECT: ¬AT(p,from) ∧ At(p,to))
  - = action schema (p, from, to need to be instantiated)
    - Action name and parameter list
    - Precondition (conj. of function-free literals)
    - Effect (conj of function-free literals and P is True and not P is false)
  - May split Add-list and delete-list in Effect

Example

- Paula flies from Philadelphia to Baltimore
  - Action(Fly(p,from,to)
  - PRECOND: At(p,from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
  - EFFECT: ¬At(p,from) ∧ At(p,to))
- We begin with
  - At(paula,phl) ∧ Plane(phl) ∧ Airport(phl) ∧ Airport(bwi)
- We take the action
  - Fly(paula, phl, bwi)
- We end with
  - ←At(paula,phl) ∧ At(paula, bwi)
- Note that we haven't said anything in the effect about what happened to the plane. Do we care?

Language semantics?

- How do actions affect states?
  - An action is applicable in any state that satisfies the precondition.
  - For FO action schema applicability involves a substitution θ for the variables in the PRECOND.
    - At(P1,JFK) ∧ At(P2,SFO) ∧ Plane(P1) ∧ Plane(P2) ∧ Airport(JFK) ∧ Airport(SFO)
    - Satisfies : At(p,from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)
    - With θ ={p/P1,from/JFK,to/SFO}
    - Thus the action is applicable.

Language semantics?

- The result of executing action a in state s is the state s’
  - s’ is same as s except
    - Any positive literal P in the effect of a is added to s’
    - Any negative literal ¬P is removed from s’
    - EFFECT: ¬AT(p,from) ∧ At(p,to):
    - At(P1,SFO) ∧ At(P2,SFO) ∧ Plane(P1) ∧ Plane(P2) ∧ Airport(JFK) ∧ Airport(SFO)
  - STRIPS assumption: (avoids representational frame problem)
    - every literal NOT in the effect remains unchanged

Planning Languages

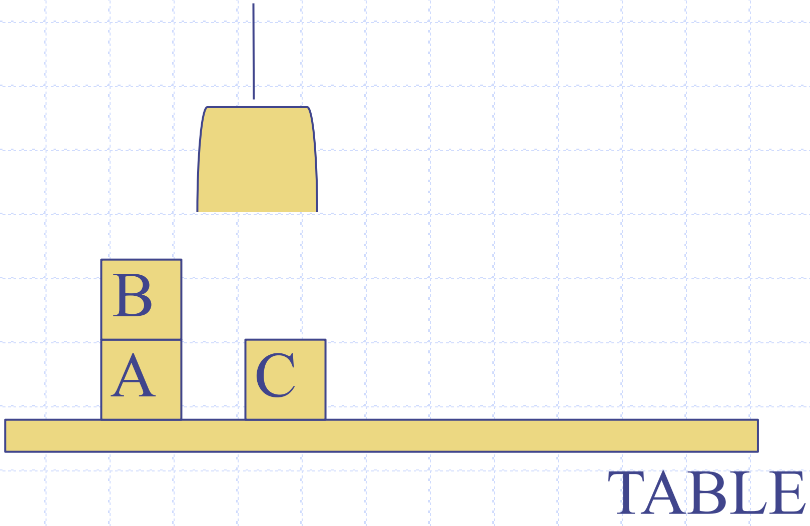
- STRIPS is simplest
  - Important limit: function-free literals
    - Allows for propositional representation
    - Function symbols lead to infinitely many states and actions
    - But poor expressivity
- Extension:Action Description language (ADL)
  - Allows negative literals
  - Allows quantified variables, conjunctions, disjunctions in goals
  - Open World assumption
    - Action(Fly(p:Plane, from: Airport, to: Airport),
    - PRECOND: At(p,from) ∧ (from ≠ to)
    - EFFECT: ¬At(p,from) ∧ At(p,to))

Planning Domain Definition Language

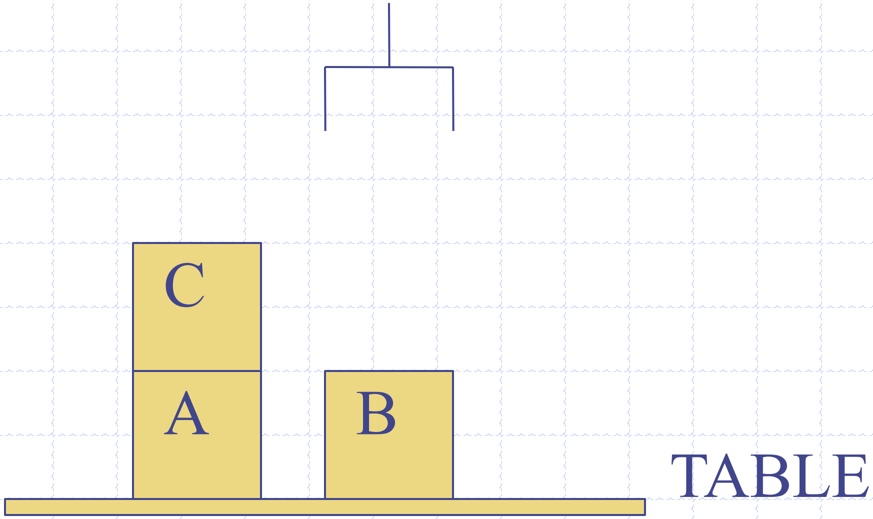
- Components:
  - Objects: things we are interested in
  - Predicates: properties of objects, true or false
  - Initial state: state of the world we start in
  - Goal specification: state we want to end up in
  - Actions: ways we can change state
- Format
  - domain file: predicates and actions
  - problem file: objects, initial state, goal

Blocks world

- The blocks world is a micro-world that consists of a table, a set of blocks and a robot hand.
- Some domain constraints:
  - Only one block can be on another block
  - Any number of blocks can be on the table
  - The hand can only hold one block
- Typical representation:
  - ontable(a)
  - ontable(c)
  - on(b,a)
  - handempty
  - clear(b)
  - clear(c)

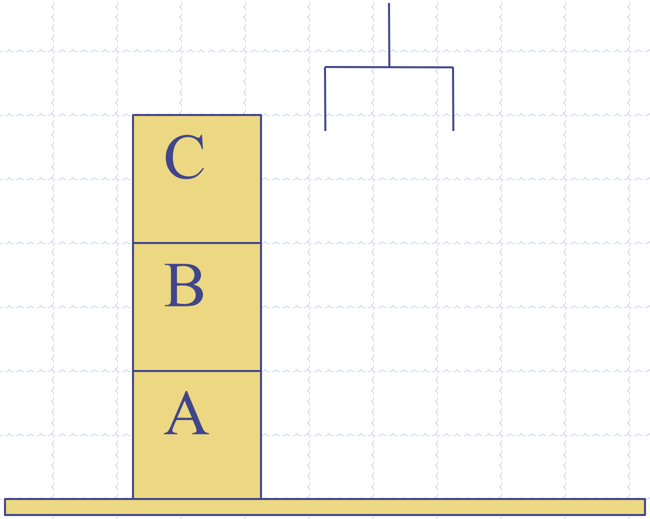


State Representation



Conjunction of propositions:
BLOCK(A), BLOCK(B), BLOCK(C),
ON(A,TABLE), ON(B,TABLE), ON(C,A),
CLEAR(B), CLEAR(C), HANDEMPTY

Goal Representation



Conjunction of propositions:
ON(A,TABLE), ON(B,A), ON(C,B)

The goal G is achieved in a state S if all the propositions in G are also in S

Action Representation

Unstack(x,y)

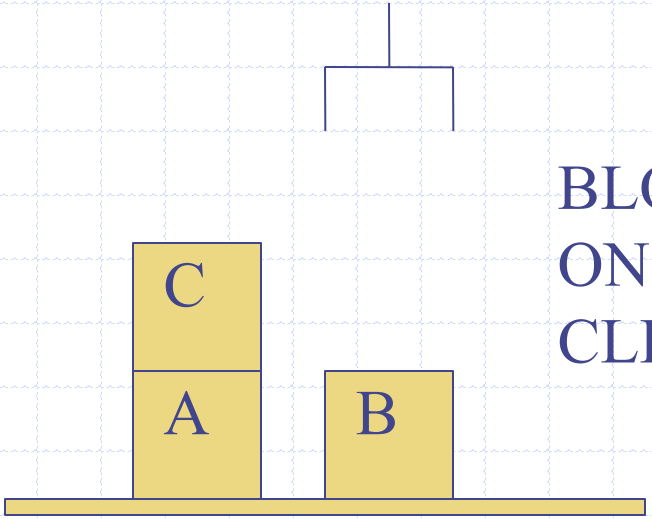
P = HANDEMPTY, BLOCK(x), BLOCK(y),
CLEAR(x), ON(x,y)

E = ¬HANDEMPTY, ¬CLEAR(x), HOLDING(x),
¬ ON(x,y), CLEAR(y)

Effect: list of literals

Precondition: conjunction of propositions

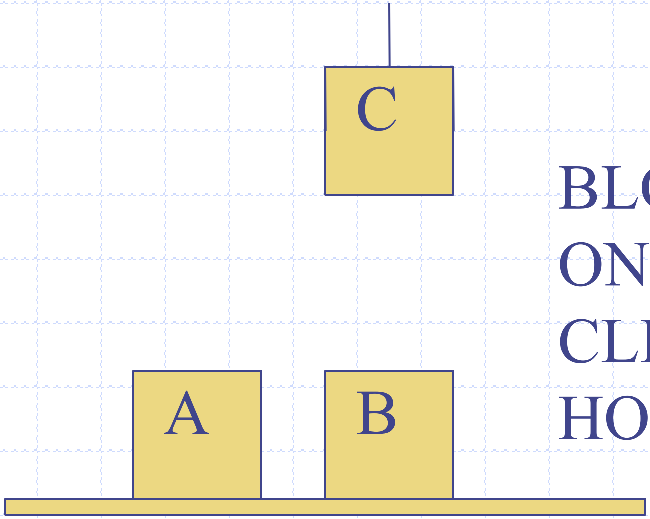
Example



BLOCK(A), BLOCK(B), BLOCK(C),
ON(A,TABLE), ON(B,TABLE), ON(C,A),
CLEAR(B), CLEAR(C), HANDEMPTY

Unstack(C,A)
P = HANDEMPTY, BLOCK(C), BLOCK(A),
CLEAR(C), ON(C,A)
E = ¬HANDEMPTY, ¬CLEAR(C), HOLDING(C),
¬ ON(C,A), CLEAR(A)

Example



BLOCK(A), BLOCK(B), BLOCK(C),
ON(A,TABLE), ON(B,TABLE), ~~ON(C,A)~~,
CLEAR(B), ~~CLEAR(C)~~, ~~HANDEMPTY~~,
HOLDING(C), CLEAR(A)

Unstack(C,A)
P = HANDEMPTY, BLOCK(C), BLOCK(A),
CLEAR(C), ON(C,A)
E = ¬HANDEMPTY, ¬CLEAR(C), HOLDING(C),
¬ ON(C,A), CLEAR(A)

Action Representation

Action(Unstack(x,y)
P: HANDEMPTY, BLOCK(x), BLOCK(y), CLEAR(x), ON(x,y)
E: ¬HANDEMPTY, ¬CLEAR(x), HOLDING(x), ¬ ON(x,y), CLEAR(y)

Action(Stack(x,y)
P: HOLDING(x), BLOCK(x), BLOCK(y), CLEAR(y)
E: ON(x,y), ¬CLEAR(y), ¬HOLDING(x), CLEAR(x), HANDEMPTY

Actions

Action(Pickup(x)
P: HANDEMPTY, BLOCK(x), CLEAR(x), ON(x,TABLE)
E: ¬HANDEMPTY, ¬CLEAR(x), HOLDING(x), ¬ON(x,TABLE)

Action(PutDown(x)
P: HOLDING(x)
E: ON(x,TABLE), ¬HOLDING(x), CLEAR(x), HANDEMPTY

Example: Spare tire problem

Init(At(Flat, Axle) ∧ At(Spare,trunk))

Goal(At(Spare,Axle))

Action(Remove(Spare,Trunk)

PRECOND: At(Spare,Trunk)

EFFECT: ¬At(Spare,Trunk) ∧ At(Spare,Ground))

Action(Remove(Flat,Axle)

PRECOND: At(Flat,Axle)

EFFECT: ¬At(Flat,Axle) ∧ At(Flat,Ground))

Action(PutOn(Spare,Axle)

PRECOND: At(Spare,Groundp) ∧¬At(Flat,Axle)

EFFECT: At(Spare,Axle) ∧ ¬At(Spare,Ground))

Action(LeaveOvernight

PRECOND:

EFFECT: ¬ At(Spare,Ground) ∧ ¬ At(Spare,Axle) ∧ ¬ At(Spare,trunk) ∧ ¬ At(Flat,Ground) ∧ ¬ At(Flat,Axle))

This example is ADL: negative literal in pre-condition

State-Space Search

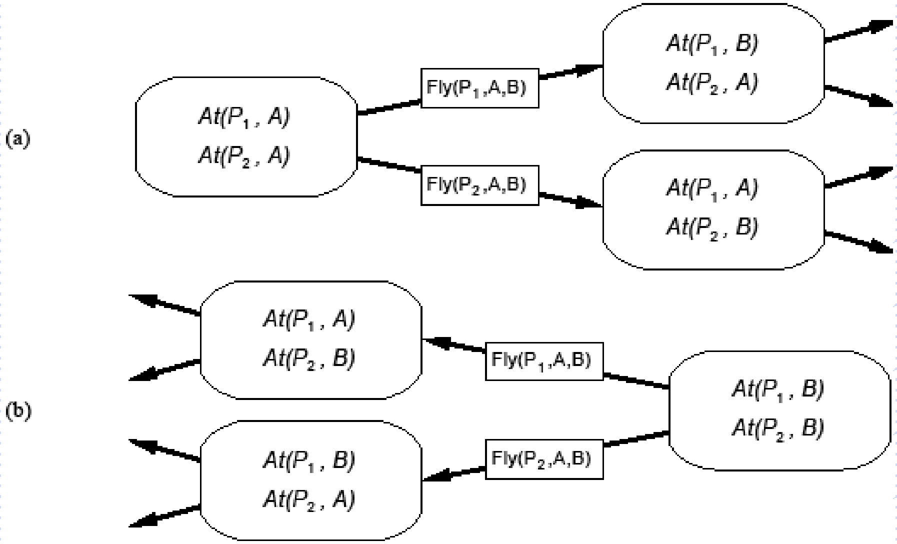
- Search the space of states (first chapters)
  - Initial state, goal test, step cost, etc.
  - Actions are the transitions between state
- Actions are invertible (why?)
  - Move forward from the initial state: Forward State-Space Search or Progression Planning
  - Move backward from goal state: Backward State-Space Search or Regression Planning

State-Space Formulation

- Formulation as state-space search problem:
  - Initial state = initial state of the planning problem
    - Literals not appearing are false
  - Actions = those whose preconditions are satisfied
    - Add positive effects, delete negative
  - Goal test = does the state satisfy the goal?
  - Step cost = each action costs 1
  - Solution is a sequence of actions.

SRIPS in State-Space Search

- STRIPS representation makes it easy to focus on 'relevant' propositions and
  - Work backward from goal (using EFFECTS)
  - Work forward from initial state (using PRECONDITIONS)
  - Facilitating bidirectional search



Relevant Action

- An action is relevant
  - In Progression planning when its preconditions match a subset of the current state
  - In Regression planning, when its effects match a subset of the current goal state

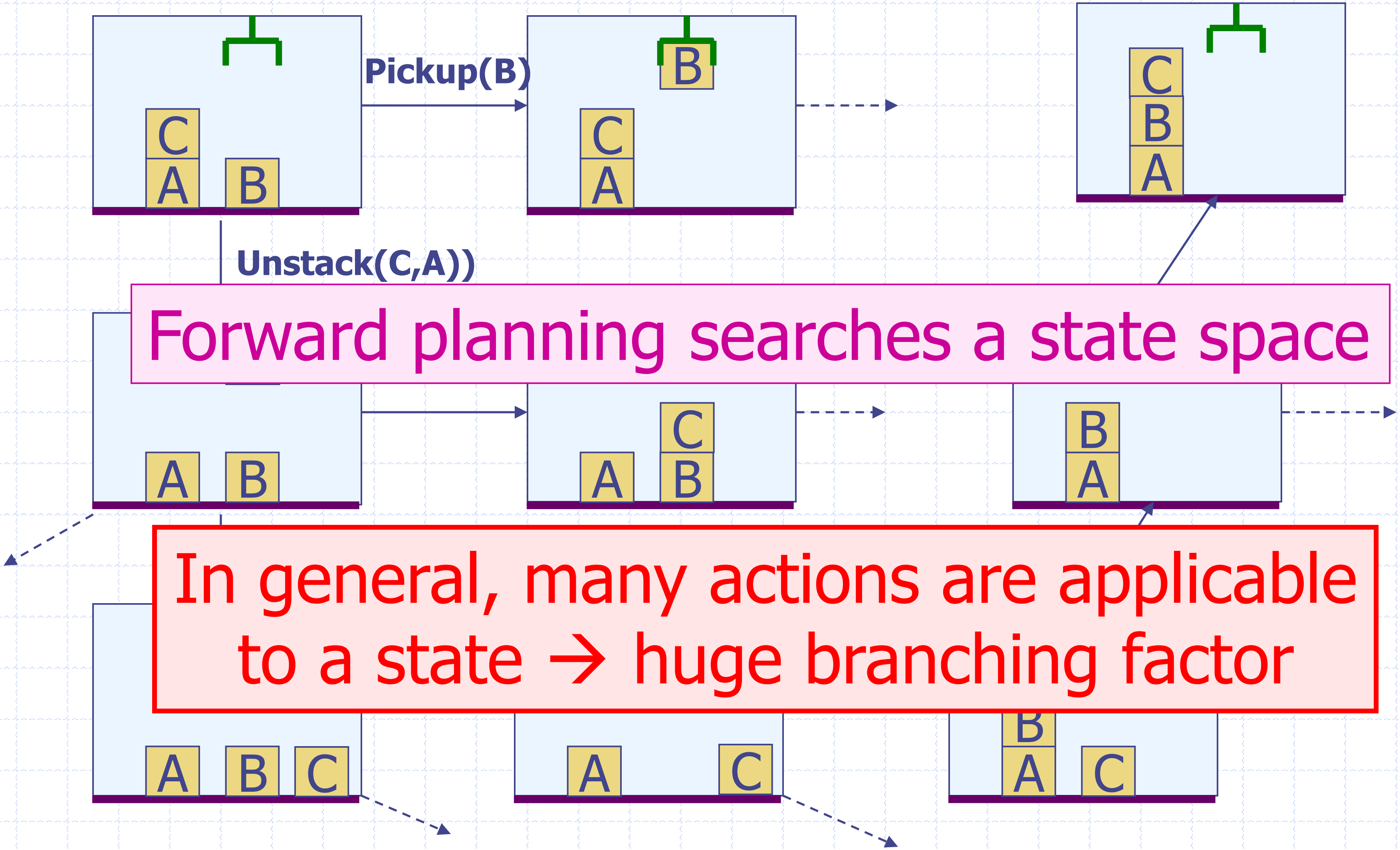
Consistent Action

- The purpose of applying an action is to ‘achieves a desired literal’
- We should be careful that the action does not undo a desired literal (as a side effect)
- A consistent action is an action that does not undo a desired literal

Progression Algorithm

- No functions, so the number of states is finite … any graph search that is complete is a complete planning algorithm.
  - E.g. A*
- Inefficient:
  - (1) irrelevant action problem
  - (2) good heuristic required for efficient search

Forward Planning



Forward planning searches a state space

In general, many actions are applicable to a state → huge branching factor

Regression algorithm

- How to determine predecessors?
  - What are the states from which applying a given action leads to the goal?
- 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 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 added to perform the search.
- Termination when predecessor satisfied by initial state.
  - In FO case, satisfaction might require a substitution.

Computation of R[G,A]

```
1. If any element of G is deleted by A
   then return False
2. G' ← Precondition of A
3. For every element SG of G do
      If SG is not added by A then add SG to G'
4. Return G'
```

Example

- G = CLEAR(B), ON(C,B)
- Stack(C,B)
  - P = HOLDING(C), BLOCK(C), BLOCK(B), CLEAR(B)
  - E = ON(C,B), ¬CLEAR(B), ¬HOLDING(C), CLEAR(C), HANDEMPTY
- R[G,Stack(C,B)] = False

Example

- G = ON(B,A), ON(C,B)
- Stack(C,B)
  - P = HOLDING(C), BLOCK(C), BLOCK(B), CLEAR(B)
  - E = ON(C,B), ¬CLEAR(B), ¬HOLDING(C), CLEAR(C), HANDEMPTY
- R[G,Stack(C,B)] = ON(B,A), HOLDING(C), BLOCK(C), BLOCK(B), CLEAR(B)

Inconsistent Regression

- G = ON(B,A), ON(C,B)

also called state constraints

- Stack(B
  - P = HO
  - E = ON
    CLE
- R[G,Sta

Inconsistency rules:

$\text{HOLDING}(x) \land \text{ON}(y,x) \Rightarrow \text{False}$

$\text{HOLDING}(x) \land \text{ON}(x,y) \Rightarrow \text{False}$

$\text{HOLDING}(x) \land \text{HOLDING}(y) \Rightarrow \text{False}$

Etc…

HOLDING(B), BLOCK(A), CLEAR(A), ON(C,B)

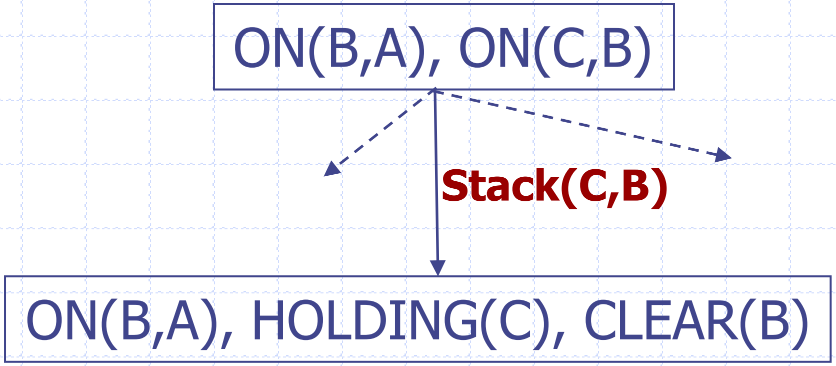
impossible

Computation of R[G,A]

1. If any element of G is deleted by A then return False
2. G′ ← Precondition of A
3. For every element SG of G do

   If SG is not added by A then add SG to G′
4. If an inconsistency rule applies to G′ then return False
5. Return G′

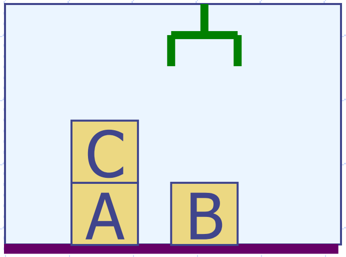
Backward Chaining



ON(B,A), ON(C,B)

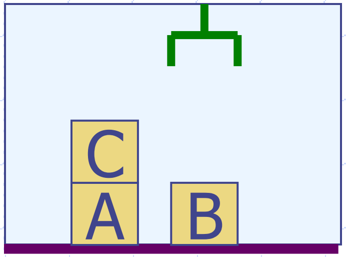
Stack(C,B)

ON(B,A), HOLDING(C), CLEAR(B)



In general, there are much fewer actions relevant to a goal than there are actions applicable → smaller branching factor than with forward planning

Backward Chaining



ON(B,A), ON(C,B)

Stack(C,B)

ON(B,A), HOLDING(C), CLEAR(B)

Pickup(C)

ON(B,A), CLEAR(B), HANDEMPTY, CLEAR(C), ON(C,TABLE)

Backward planning searches a goal space

CLEAR(C), ON(C,TABLE), HOLDING(B), CLEAR(A)

Pickup(B)

CLEAR(C), ON(C,TABLE), CLEAR(A), HANDEMPTY, CLEAR(B), ON(B,TABLE)

Putdown(C)

CLEAR(A), CLEAR(B), ON(B,TABLE), HOLDING(C)

Unstack(C,A)

CLEAR(B), ON(B,TABLE), CLEAR(C), HANDEMPTY, ON(C,A)

Heuristic to Speed up Search

- Neither progression or regression are very efficient without a good heuristic.
  - How many actions are needed to achieve the goal?
  - Exact solution is NP hard, find a good estimate
- We can use A*, but we need an admissible heuristic
  1. Divide-and-conquer: sub-goal independence assumption
  - Problem relaxation by removing
  2. … all preconditions
  3. … all preconditions <u>and</u> negative effects
  4. … negative effects only: Empty-Delete-List

1. Subgoal Independence Assumption

- The cost of solving a conjunction of subgoals is the sum of the costs of solving each subgoal independently
- Optimistic
  - Where subplans interact negatively
  - Example: one action in a subplan delete goal achieved by an action in another subplan
- Pessimistic (not admissible)
  - Redundant actions in subplans can be replaced by a single action in merged plan

2. Problem Relaxation: Removing Preconditions

- Remove preconditions from action descriptions
  - All actions are applicable
  - Every literal in the goal is achievable in one step
- Number of steps to achieve the conjunction of literals in the goal is equal to the number of unsatisfied literals
- Alert
  - Some actions may achieve several literals
  - Some action may remove the effect of another action

3. Remove Preconditions & Negative Effects

- Considers only positive interactions among actions to achieve multiple subgoals
- The minimum number of actions required is the sum of the union of the actions' positive effects that satisfy the goal
- The problem is reduced to a set cover problem, which is NP-hard
  - Approximation by a greedy algorithm cannot guarantee an admissible heuristic

4. Removing Negative Effects (Only)

- Remove all negative effects of actions (no action may destroy the effects of another)
- Known as the Empty-Delete-List heuristic
- Requires running a simple planning algorithm
- Quick & effective
- Usable in progression or regression planning

Partial-order planning

- Progression and regression planning are totally ordered plan search forms.
  - They cannot take advantage of problem decomposition.
    - Decisions must be made on how to sequence actions on all the subproblems
- Least commitment strategy:
  - Delay choice during search

Shoe example

Goal(RightShoeOn ∧ LeftShoeOn)

Init()

Action(RightShoe, PRECOND: RightSockOn
EFFECT: RightShoeOn)

Action(RightSock, PRECOND:
EFFECT: RightSockOn)

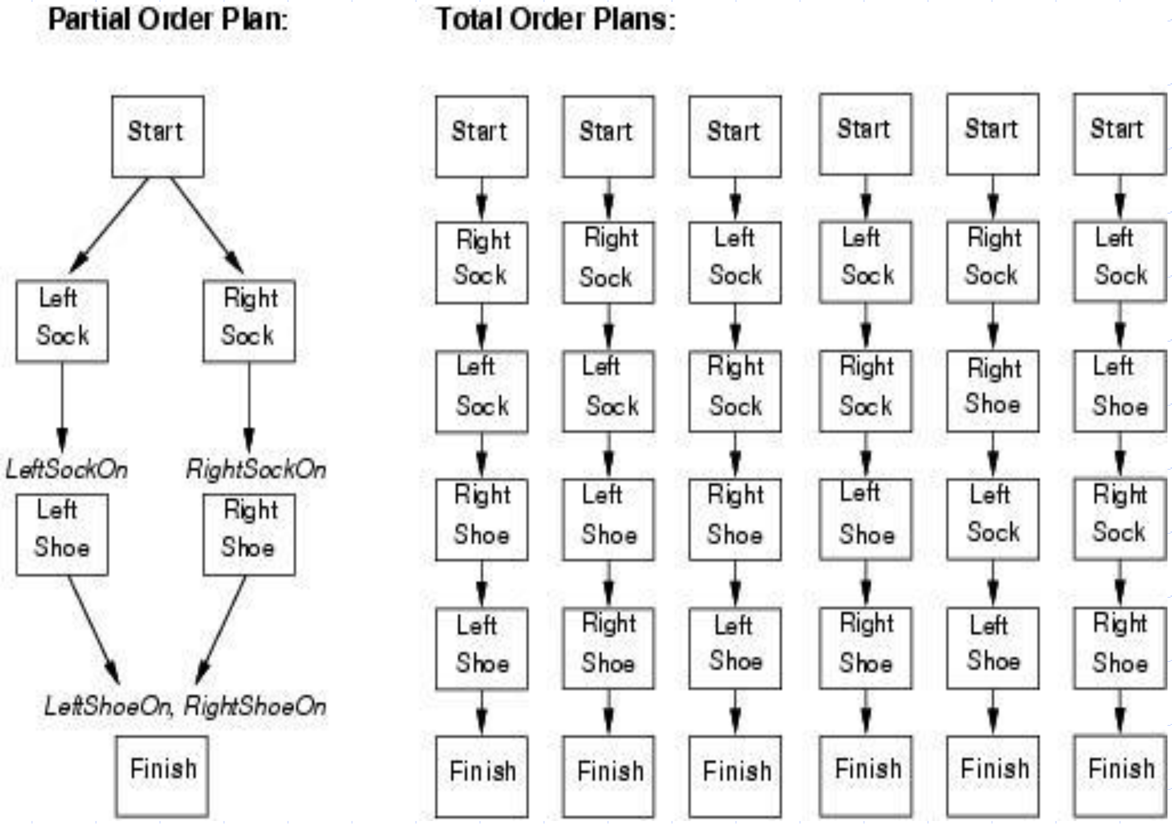
Action(LeftShoe, PRECOND: LeftSockOn
EFFECT: LeftShoeOn)

Action(LeftSock, PRECOND:
EFFECT: LeftSockOn)

Planner: combine two action sequences (1)leftsock, leftshoe (2)rightsock, rightshoe

Partial-order planning(POP)

- Any planning algorithm that can place two actions into a plan without which comes first is a PO plan.



Partial Order Plan:

Start → Left Sock → LeftSockOn → Left Shoe → LeftShoeOn, RightShoeOn → Finish

Start → Right Sock → RightSockOn → 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

Components of a Plan

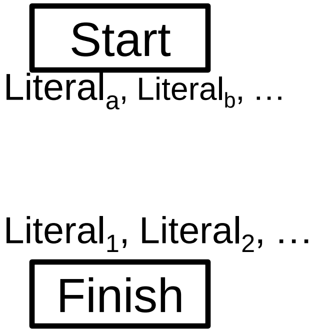
1. A set of actions
2. A set of ordering constraints
   - $A \prec B$ reads “A before B” but not necessarily immediately before B
   - Alert: caution to cycles $A \prec B$ and $B \prec A$
3. A set of causal links (protection intervals) between actions
   - $A \xrightarrow{p} B$ reads “A achieves *p* for B” and p must remain true from the time A is applied to the time B is applied
   - Example “RightSock $\xrightarrow{\textit{RightSockOn}}$ RightShoe
4. A set of open preconditions
   - Planners work to reduce the set of open preconditions to the empty set w/o introducing contradictions

Consistent Plan (POP)

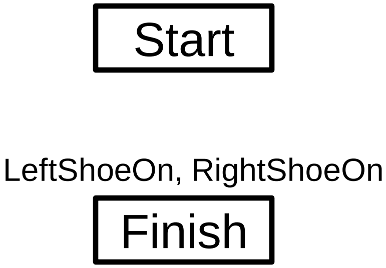
- Consistent plan is a plan that has
  - No cycle in the ordering constraints
  - No conflicts with the causal links
- Solution
  - Is a consistent plan with no open preconditions
- To solve a conflict between a causal link $A \xrightarrow{p} B$ and an action C (that clobbers, threatens the causal link), we force C to occur outside the "protection interval" by adding
  - the constraint $C \prec A$ (demoting C) or
  - the constraint $B \prec C$ (promoting C)

Setting up the PoP

- Add dummy states
  - Start
    - Has no preconditions
    - Its effects are the literals of the initial state
  - Finish
    - Its preconditions are the literals of the goal state
    - Has no effects


Start
Literal_a, Literal_b, …
Literal_1, Literal_2, …
Finish

- Initial Plan:
  - Actions: {Start, Finish}
  - Ordering constraints: {Start $\prec$ Finish}
  - Causal links: {}
  - Open Preconditions: {LeftShoeOn,RightShoeOn}



POP as a Search Problem

- The successor function arbitrarily picks one open precondition *p* on an action B
- For every possible consistent action A that achieves *p*
  - It generates a successor plan adding the causal link $A \xrightarrow{p} B$ and the ordering constraint $A \prec B$
  - If A was not in the plan, it adds $Start \prec A$ and $A \prec Finish$
  - It resolves all conflicts between
    - the new causal link and all existing actions
    - between A and all existing causal links
  - Then it adds the successor states for combination of resolved conflicts
- It repeats until no open precondition exists

Process summary

- Operators on partial plans
  - Add link from existing plan to open precondition.
  - Add a step to fulfill an open condition.
  - Order one step w.r.t 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 irresolvable.

Example: Flat tire problem

Init(At(Flat, Axle) ∧ At(Spare,trunk))

Goal(At(Spare,Axle))

Action(Remove(Spare,Trunk)

PRECOND: At(Spare,Trunk)

EFFECT: ¬At(Spare,Trunk) ∧ At(Spare,Ground))

Action(Remove(Flat,Axle)

PRECOND: At(Flat,Axle)

EFFECT: ¬At(Flat,Axle) ∧ At(Flat,Ground))

Action(PutOn(Spare,Axle)

PRECOND: At(Spare,Groundp) ∧¬At(Flat,Axle)

EFFECT: At(Spare,Axle) ∧ ¬Ar(Spare,Ground))

Action(LeaveOvernight

PRECOND:

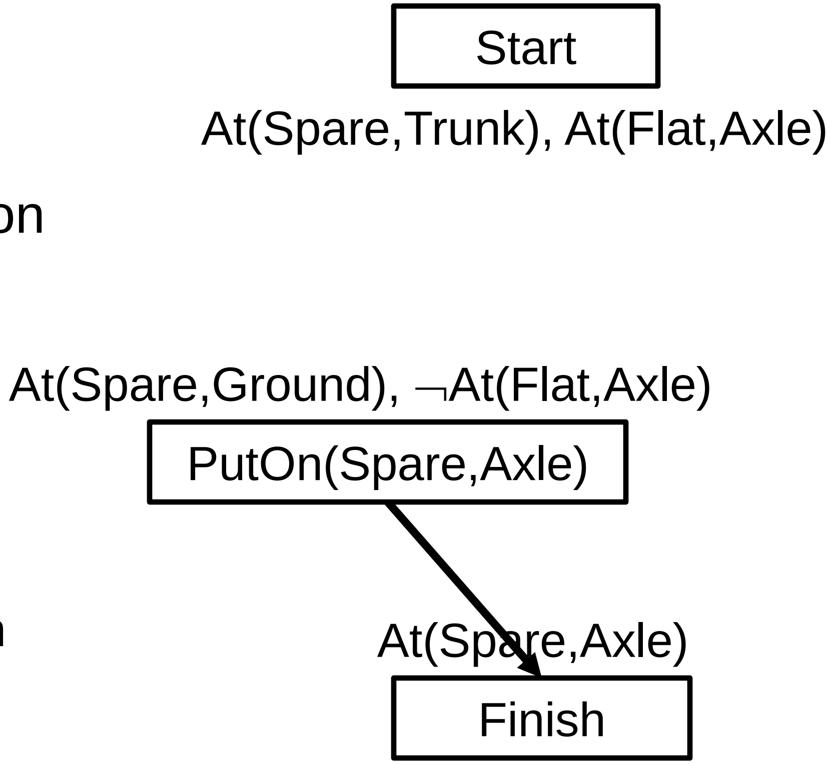
EFFECT: ¬ At(Spare,Ground) ∧ ¬ At(Spare,Axle) ∧ ¬ At(Spare,trunk) ∧ ¬ At(Flat,Ground) ∧ ¬ At(Flat,Axle))

Example of POP: Flat tire problem

- See problem description in Fig 11.7 page 391

- Only one open precondition
- Only 1 applicable action

- Pick up At(Spare,Ground)
- Choose only applicable action Remove(Spare,Trunk)



Start

At(Spare,Trunk), At(Flat,Axle)

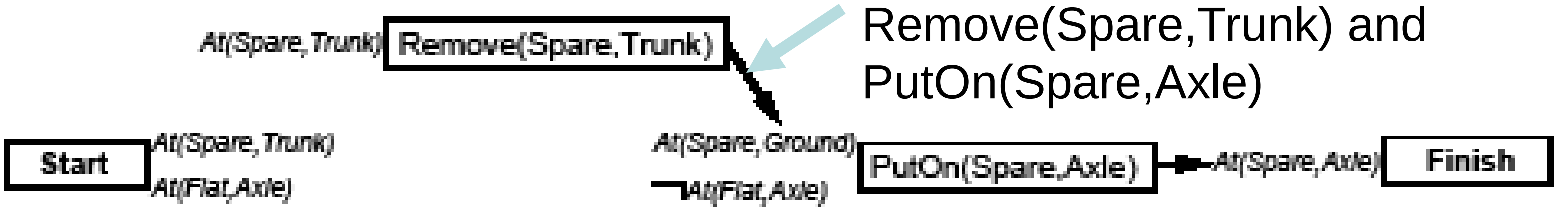
At(Spare,Ground), ¬At(Flat,Axle)

PutOn(Spare,Axle)

At(Spare,Axle)

Finish

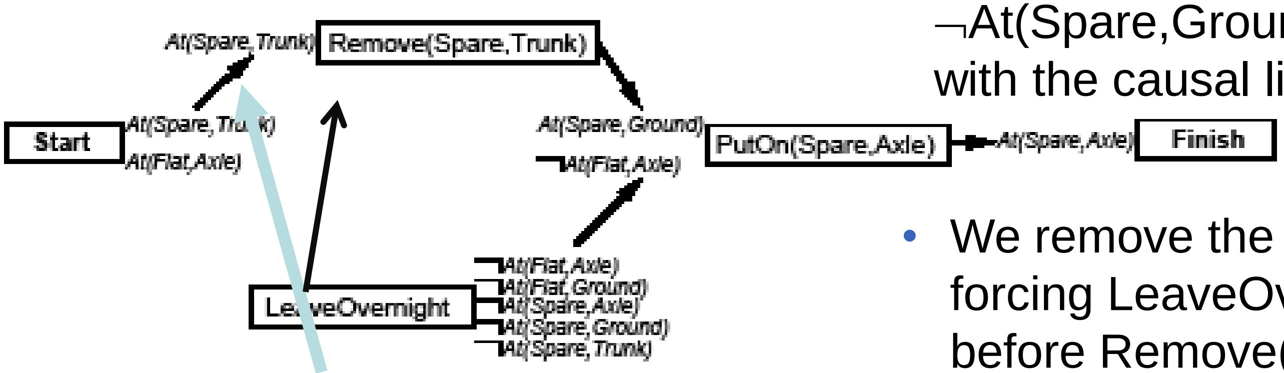
Example of POP: Flat tire problem



Add causal link between Remove(Spare,Trunk) and PutOn(Spare,Axle)

- Pick up ¬At(Flat,Axle)
- There are 2 applicable actions: LeaveOvernight and Remove(Flat,Axle)
- Choose LeaveOvernight

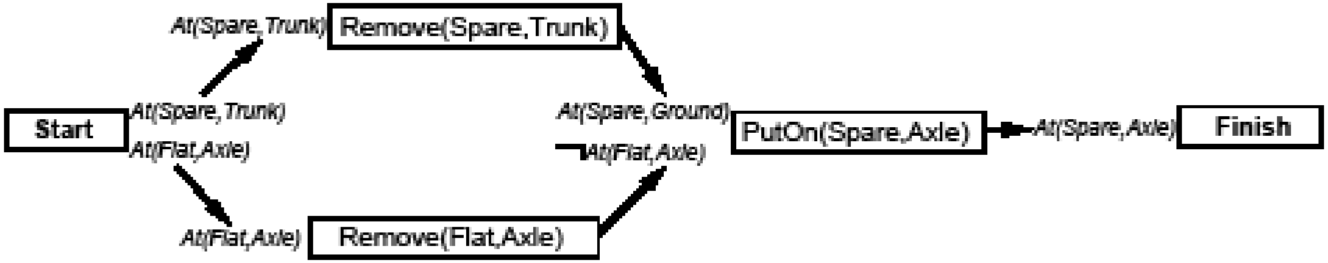
- LeaveOvernight has effect ¬At(Spare,Ground), which conflicts with the causal link



- We remove the conflict by forcing LeaveOvernight to occur before Remove(Spare,Trunk)

- Conflicts with effects of Remove(Spare,Trunk)
- The only way to resolve the conflict is to undo LeaveOvernightuse the action Remove(Flat,Axle)

Example of POP: Flat tire problem



- This time, we choose Remove(Flat,Axle)
- Pick up At(Spare,Trunk) and choose Start to achieve it
- Pick up At(Flat,Axle) and choose Start to achieve it.
- We now have a complete consistent partially ordered plan

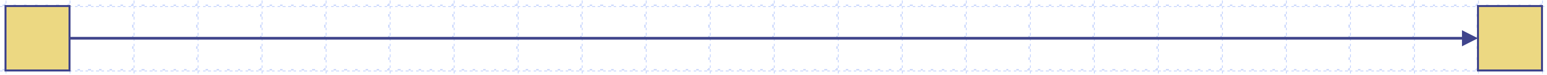
POP Algorithm (1)

- Backtrack when fails to resolve a threat or find an operator
- Causal links
  - Recognize when to abandon a doomed plan without wasting time expanding irrelevant part of the plan
  - allow early pruning of inconsistent combination of actions
- When actions include variables, we need to find appropriate substitutions
  - Typically we try to delay commitments to instantiating a variable until we have no other choice (least commitment)
- POP is sound, complete, and systematic (no repetition)

POP Algorithm (2)

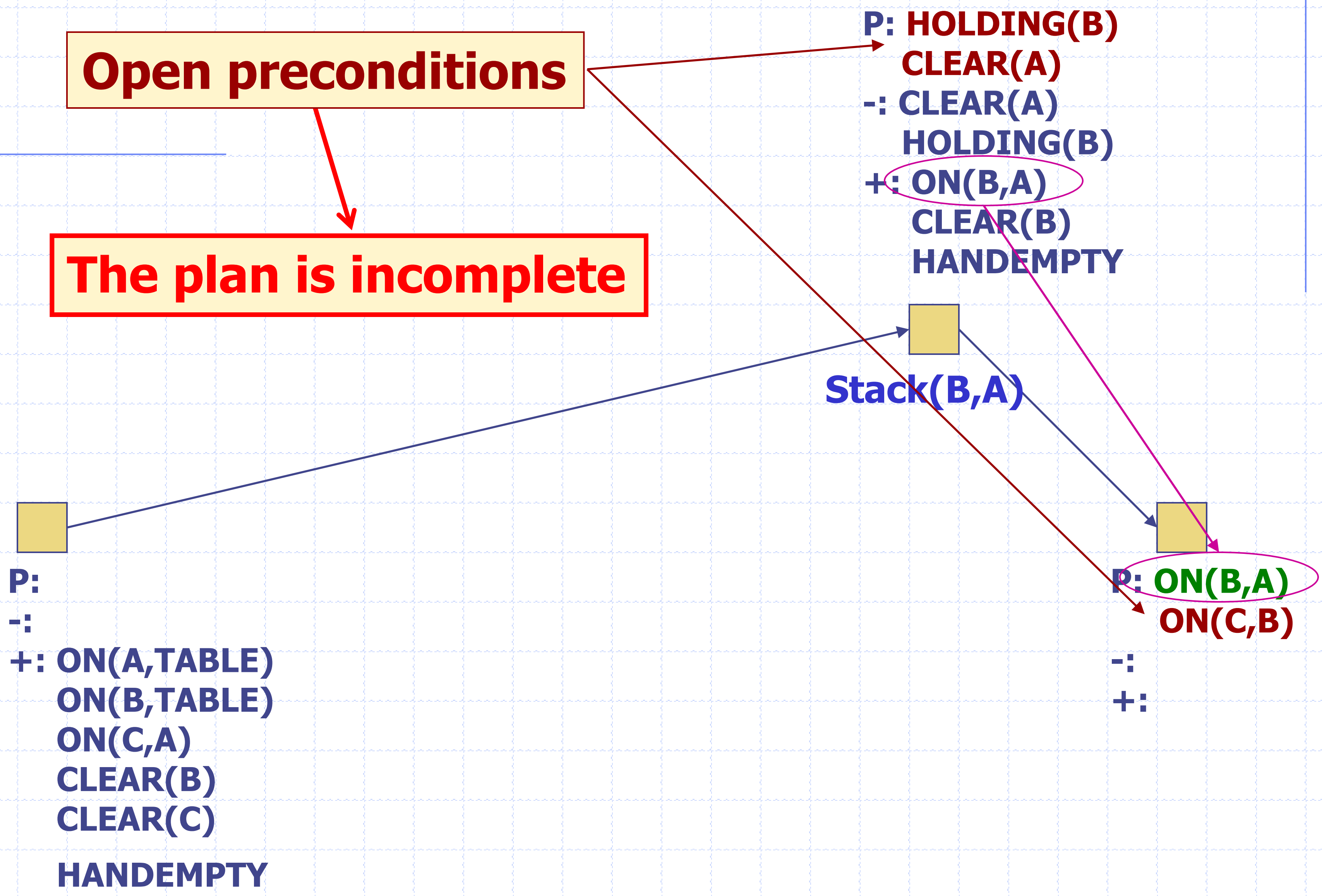
- Decomposes the problem (advantage)
- But does not represent states explicitly: it is hard to design heuristic to estimate distance from goal
  - Example: Number of open preconditions – those that match the effects of the start node. Not perfect (same problems as before)
- A heuristic can be used to choose which plan to refine (which precondition to pick-up):
  - Choose the most-constrained precondition, the one satisfied by the least number of actions. Like in CSPs!
  - When no action satisfies a precondition, backtrack!
  - When only one action satisfies a precondition, pick up the precondiction.

POP – Block Example



P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY

P: ON(B,A)
ON(C,B)
-:
+:



Open preconditions

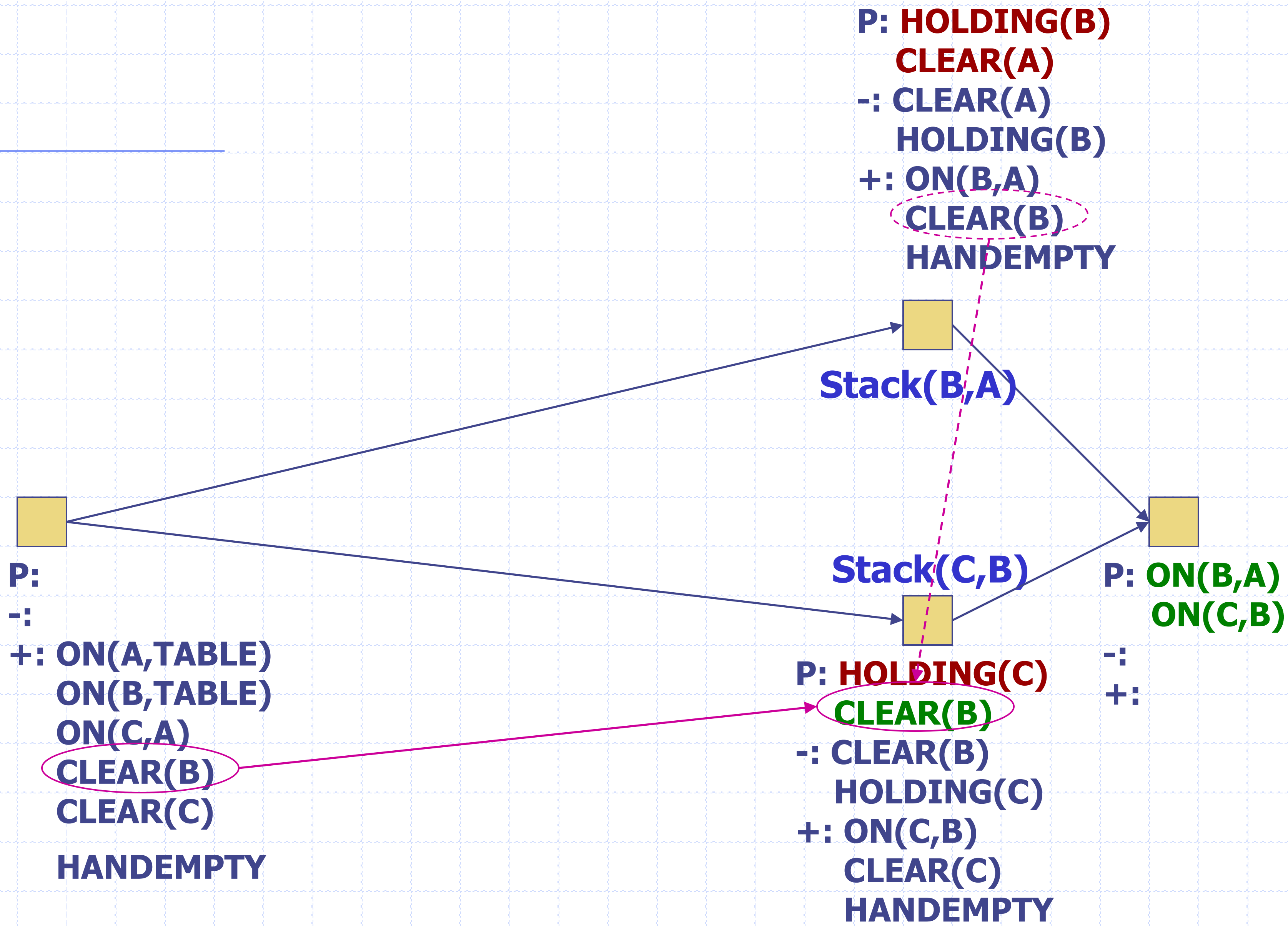
The plan is incomplete

P: HOLDING(B)
CLEAR(A)
-: CLEAR(A)
HOLDING(B)
+: ON(B,A)
CLEAR(B)
HANDEMPTY

Stack(B,A)

P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY

P: ON(B,A)
ON(C,B)
-:
+:



P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY

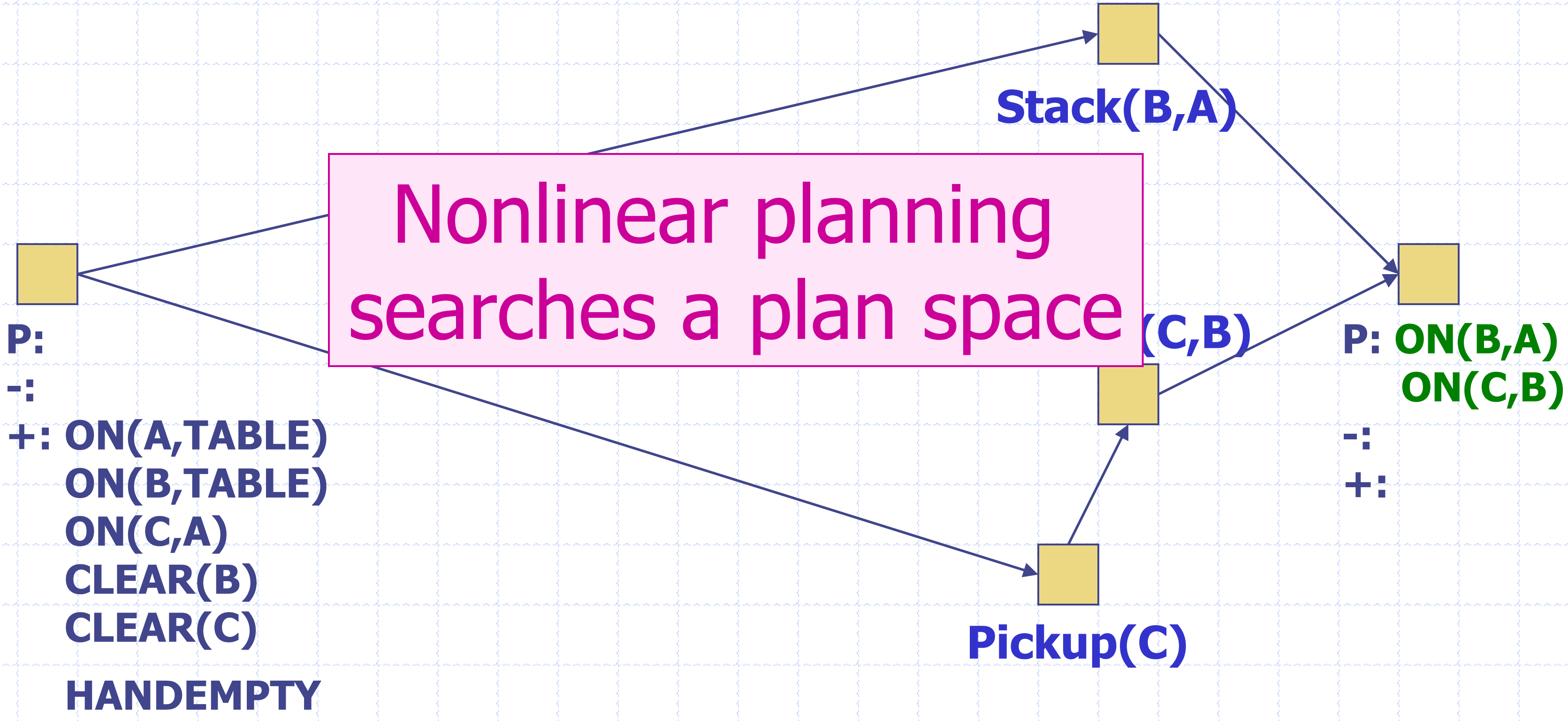
Stack(B,A)

P: HOLDING(B)
CLEAR(A)
-: CLEAR(A)
HOLDING(B)
+: ON(B,A)
CLEAR(B)
HANDEMPTY

Stack(C,B)

P: HOLDING(C)
CLEAR(B)
-: CLEAR(B)
HOLDING(C)
+: ON(C,B)
CLEAR(C)
HANDEMPTY

P: ON(B,A)
ON(C,B)
-:
+:



Nonlinear planning searches a plan space

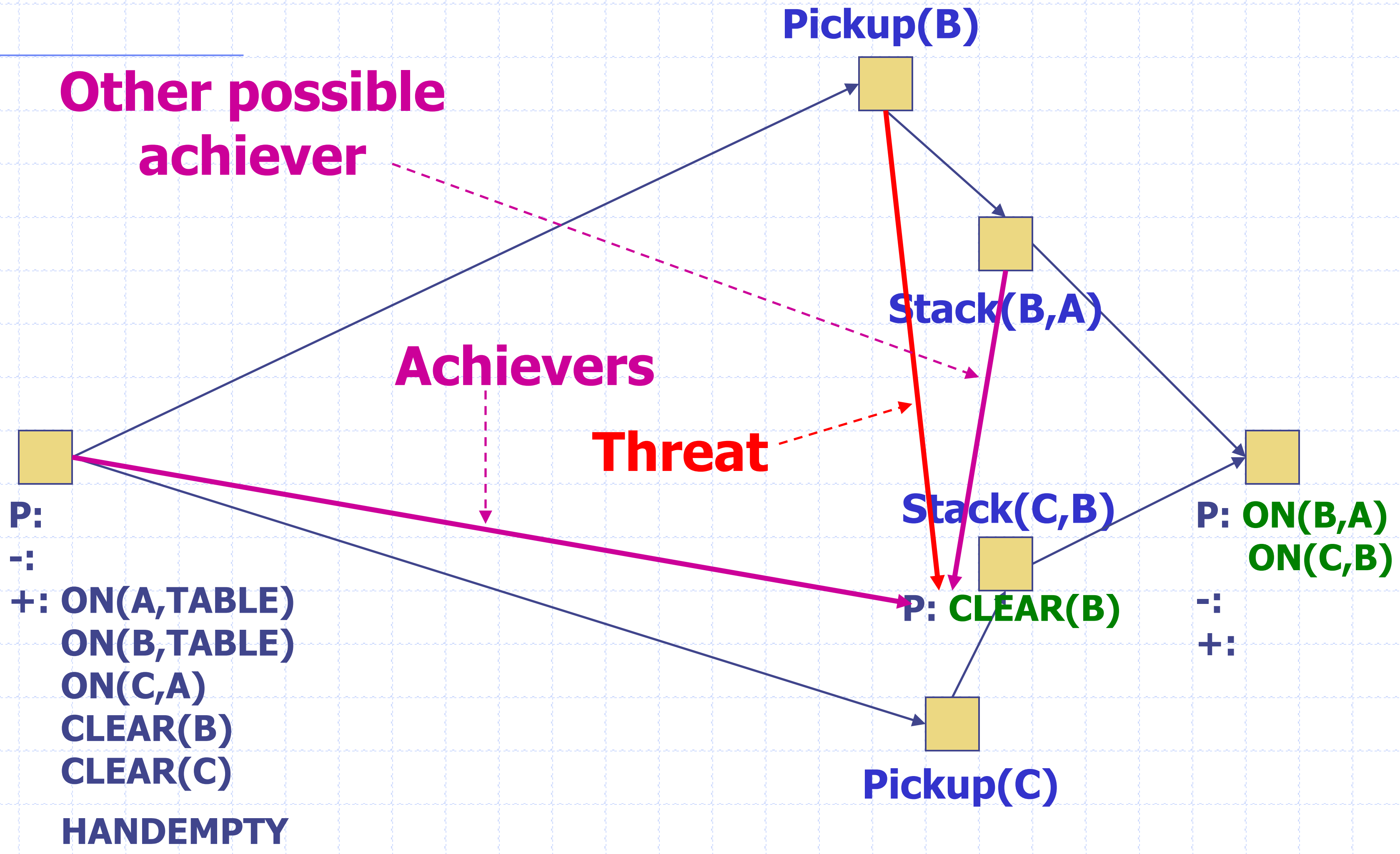
Stack(B,A)

(C,B)

Pickup(C)

P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY

P: ON(B,A)
ON(C,B)
-:
+:



Other possible achiever

Pickup(B)

Stack(B,A)

Achievers

Threat

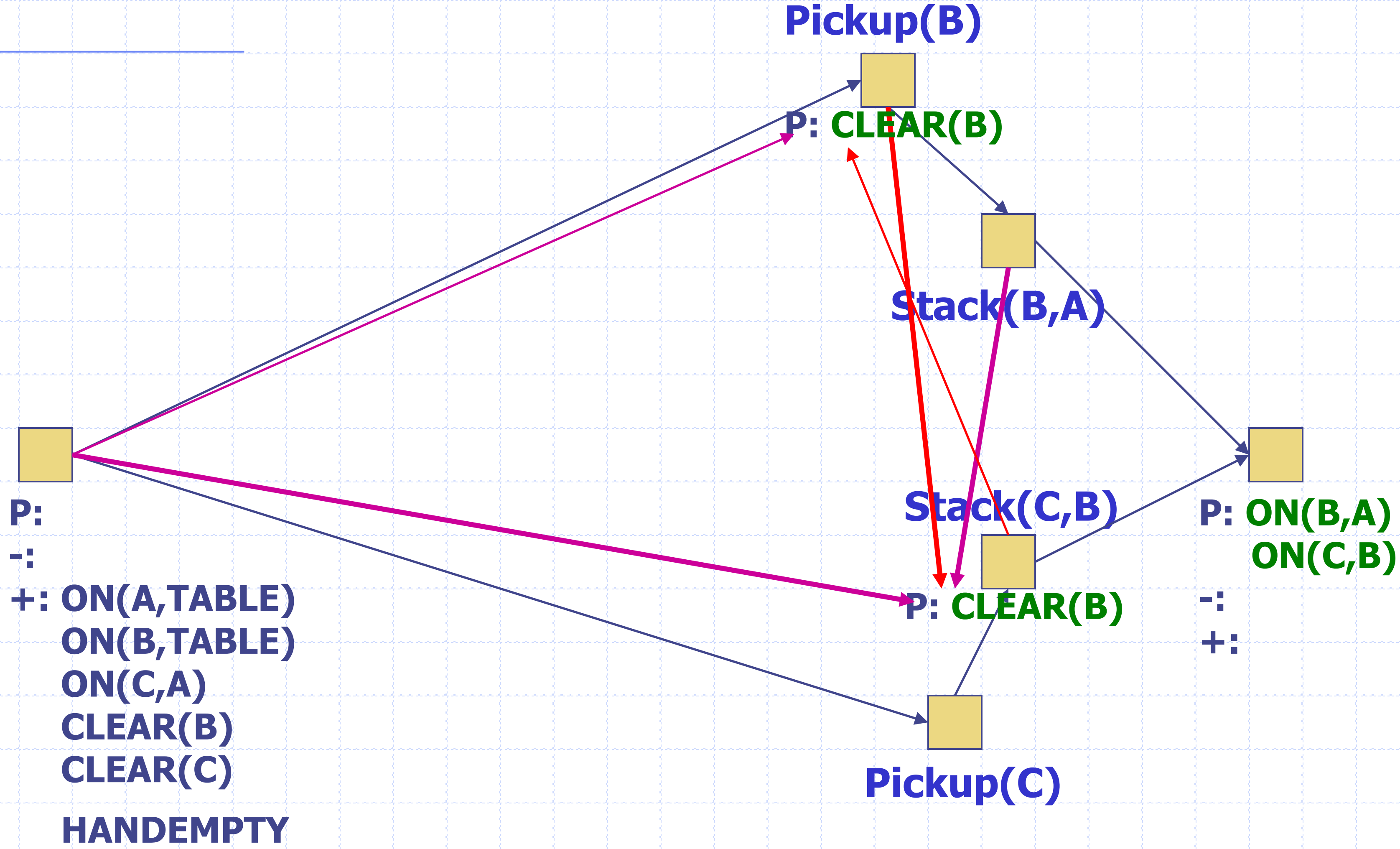
P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY

Stack(C,B)

P: CLEAR(B)

P: ON(B,A)
ON(C,B)
-:
+:

Pickup(C)



Pickup(B)

P: CLEAR(B)

Stack(B,A)

P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)

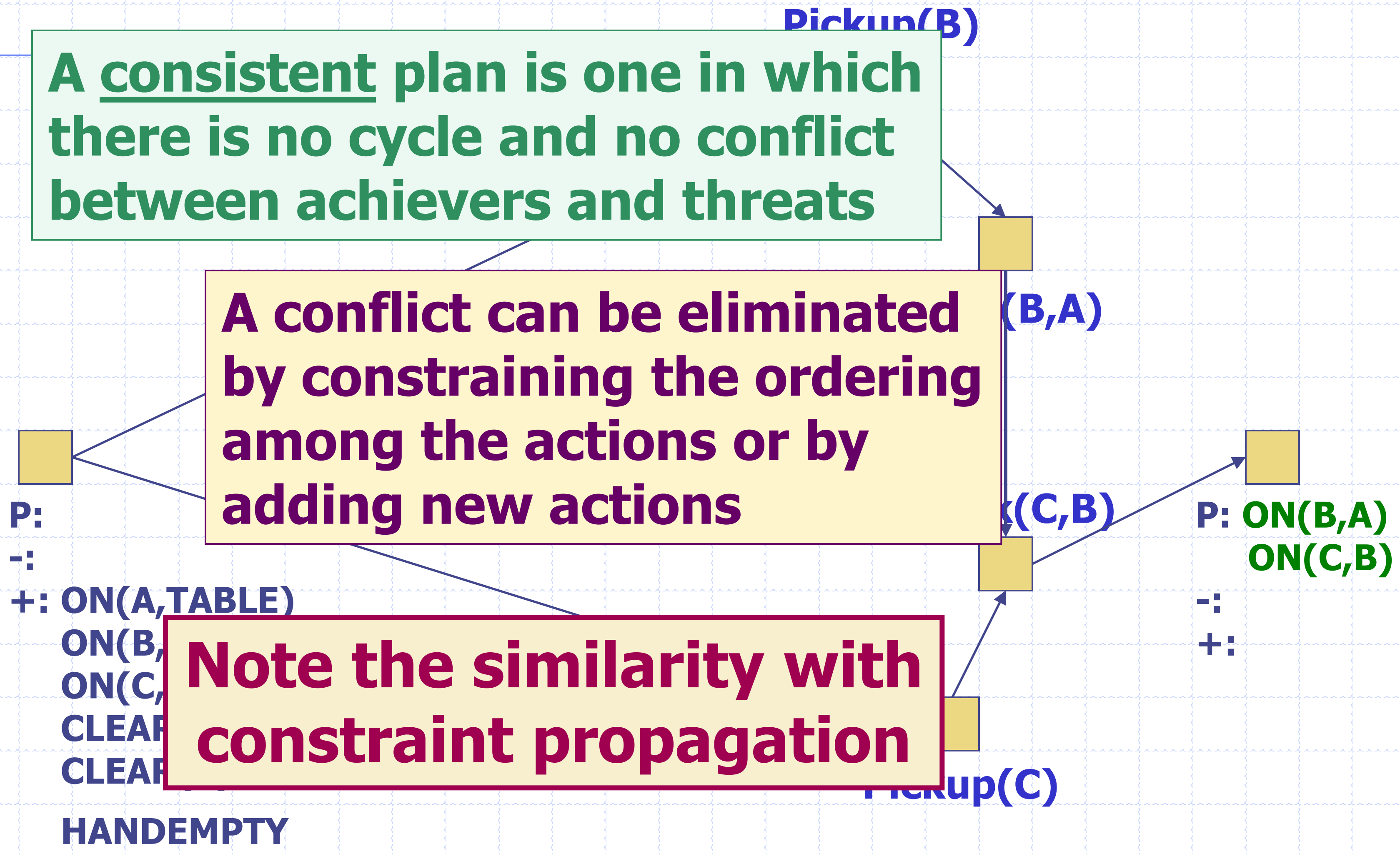
HANDEMPTY

Stack(C,B)

P: CLEAR(B)

P: ON(B,A)
ON(C,B)
-:
+:

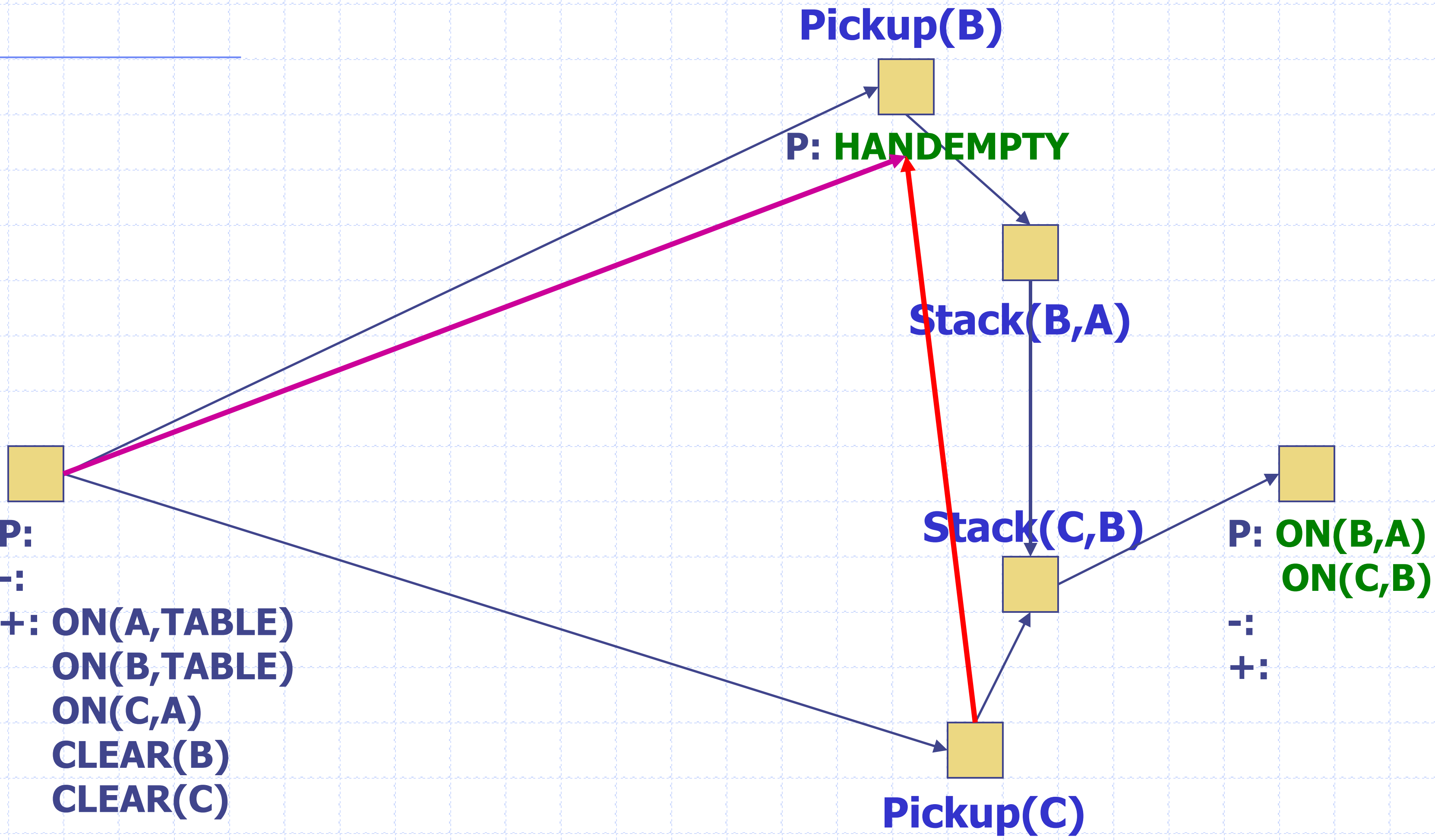
Pickup(C)



A consistent plan is one in which there is no cycle and no conflict between achievers and threats

A conflict can be eliminated by constraining the ordering among the actions or by adding new actions

Note the similarity with constraint propagation



Pickup(B)

P: HANDEMPTY

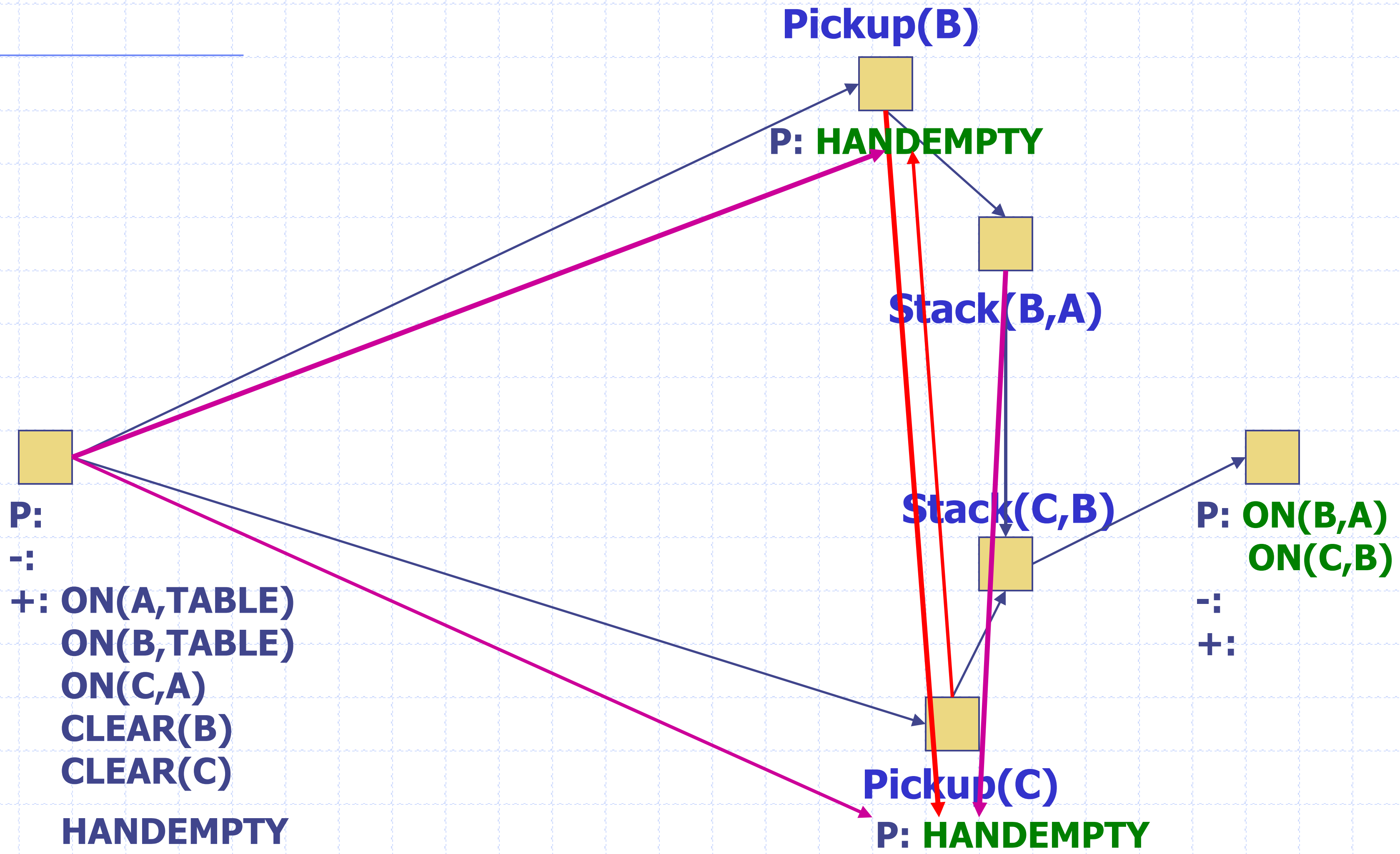
Stack(B,A)

Stack(C,B)

Pickup(C)

P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY

P: ON(B,A)
ON(C,B)
-:
+:



P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)

HANDEMPTY

Pickup(B)

P: HANDEMPTY

Stack(B,A)

Stack(C,B)

Pickup(C)

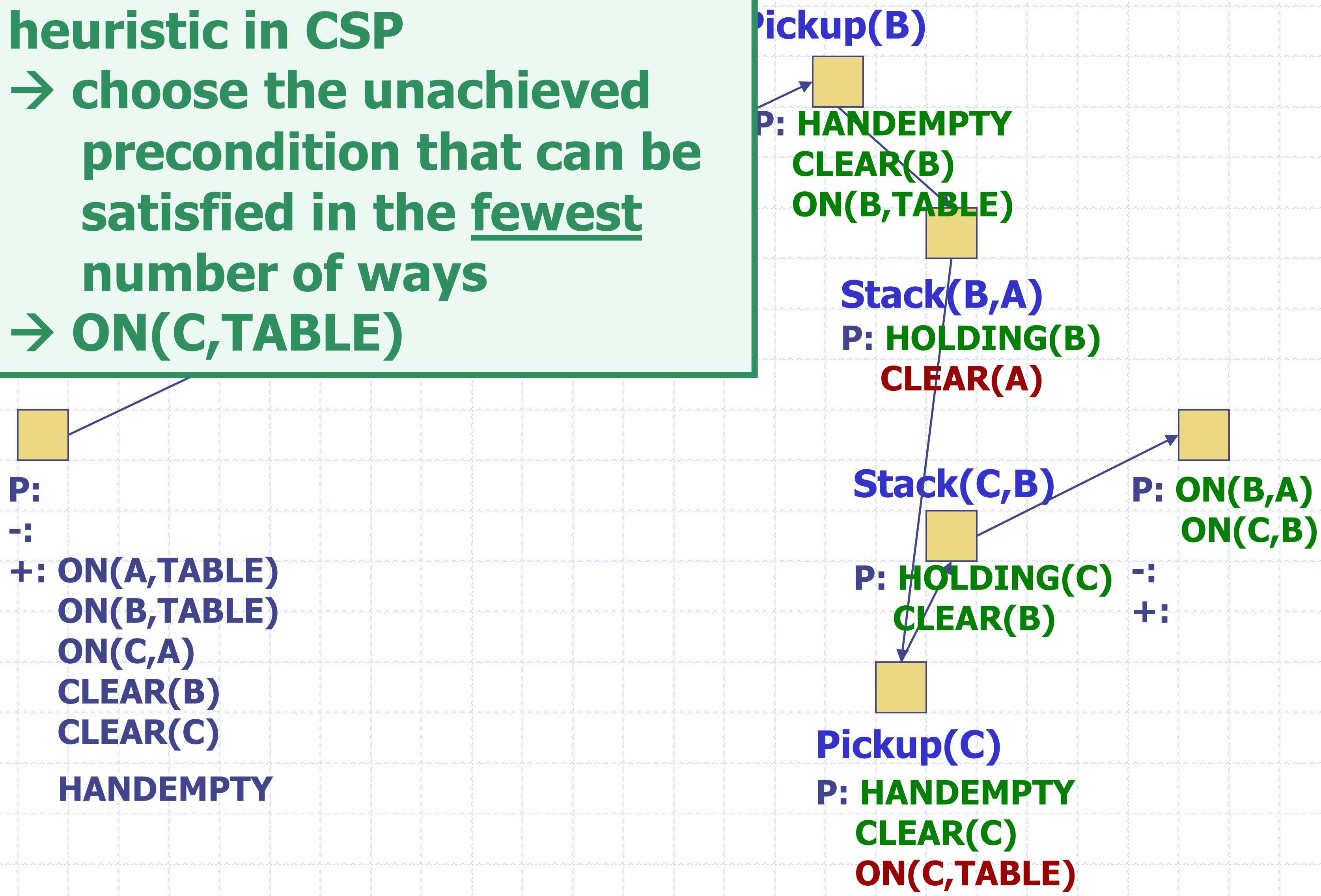
P: HANDEMPTY

P: ON(B,A)
ON(C,B)
-:
+:

~ Most-constrained-variable heuristic in CSP

→ choose the unachieved precondition that can be satisfied in the fewest number of ways

→ ON(C,TABLE)



P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY

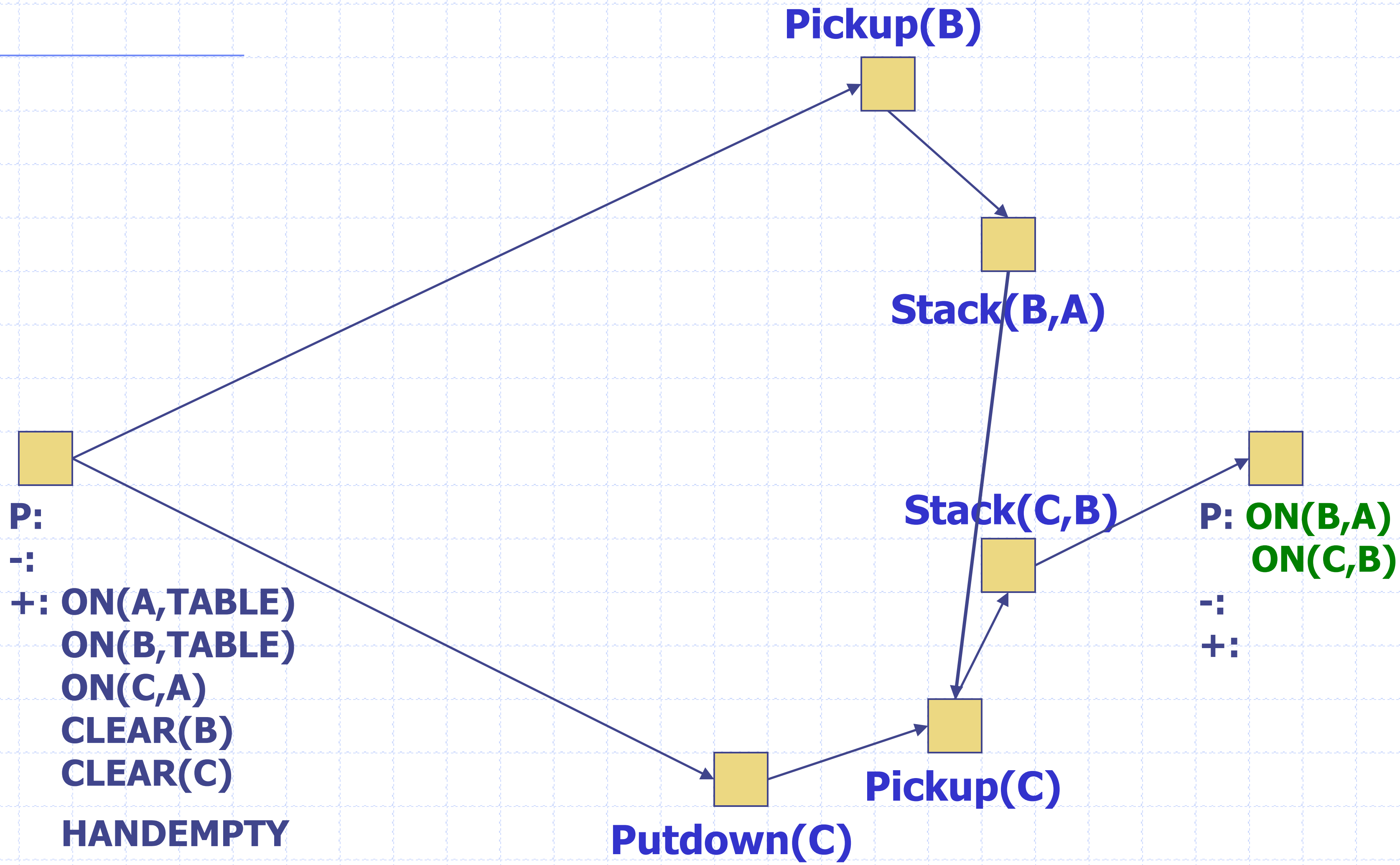
Pickup(B)
P: HANDEMPTY
CLEAR(B)
ON(B,TABLE)

Stack(B,A)
P: HOLDING(B)
CLEAR(A)

Stack(C,B)
P: HOLDING(C)
CLEAR(B)

Pickup(C)
P: HANDEMPTY
CLEAR(C)
ON(C,TABLE)

P: ON(B,A)
ON(C,B)
-:
+:



P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)

HANDEMPTY

Pickup(B)

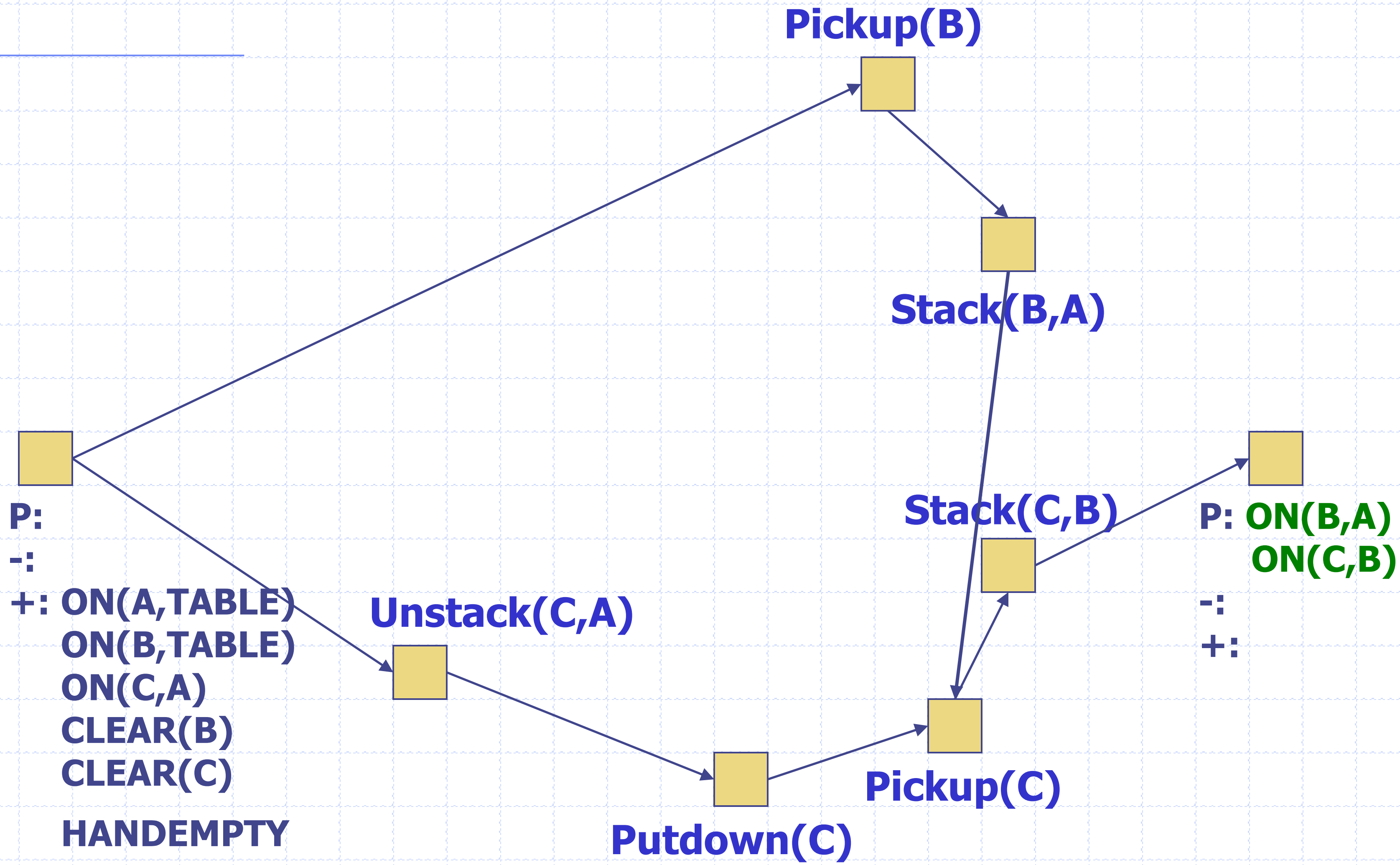
Stack(B,A)

Stack(C,B)

Pickup(C)

Putdown(C)

P: ON(B,A)
ON(C,B)
-:
+:



P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)

HANDEMPTY

Pickup(B)

Stack(B,A)

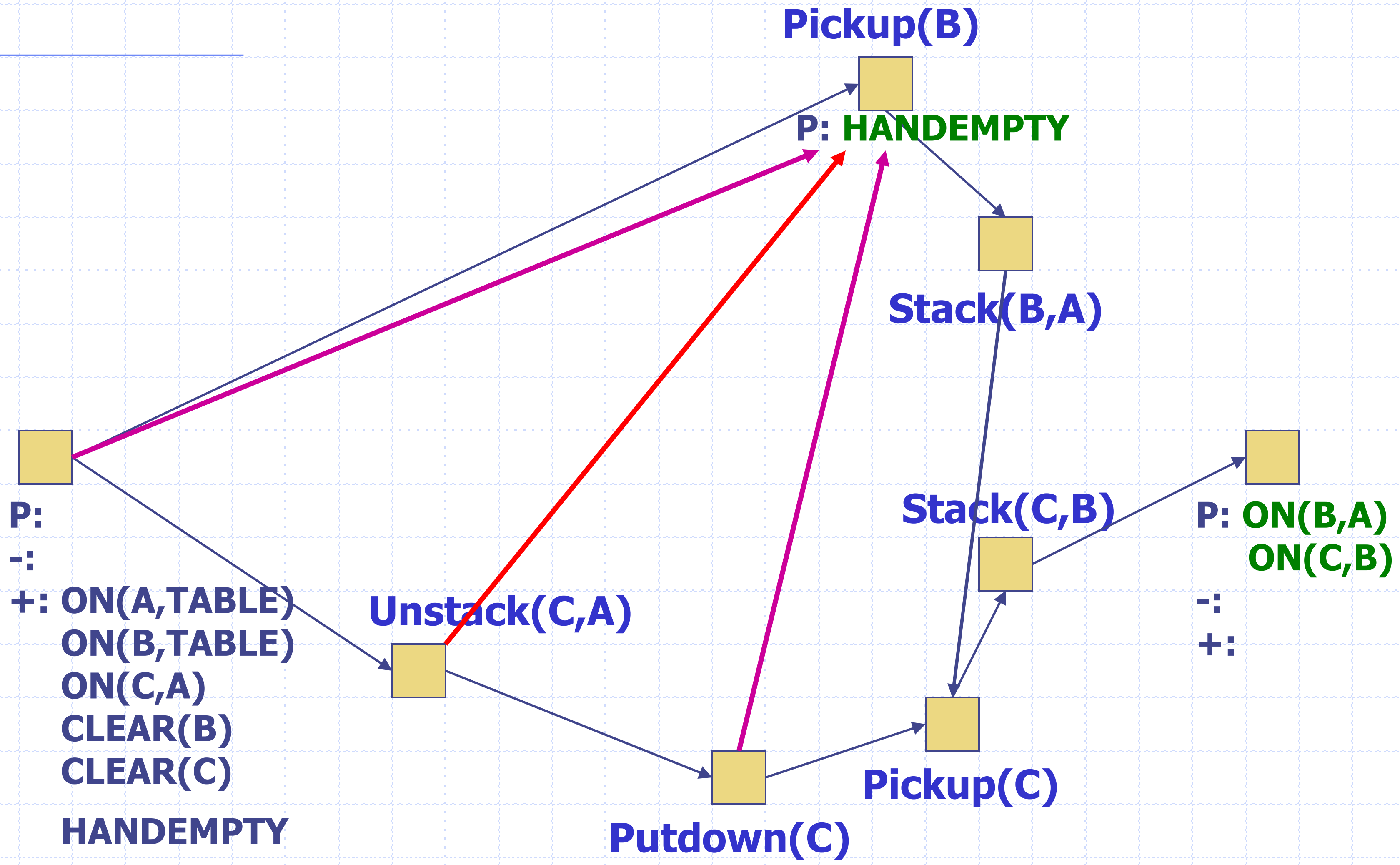
Unstack(C,A)

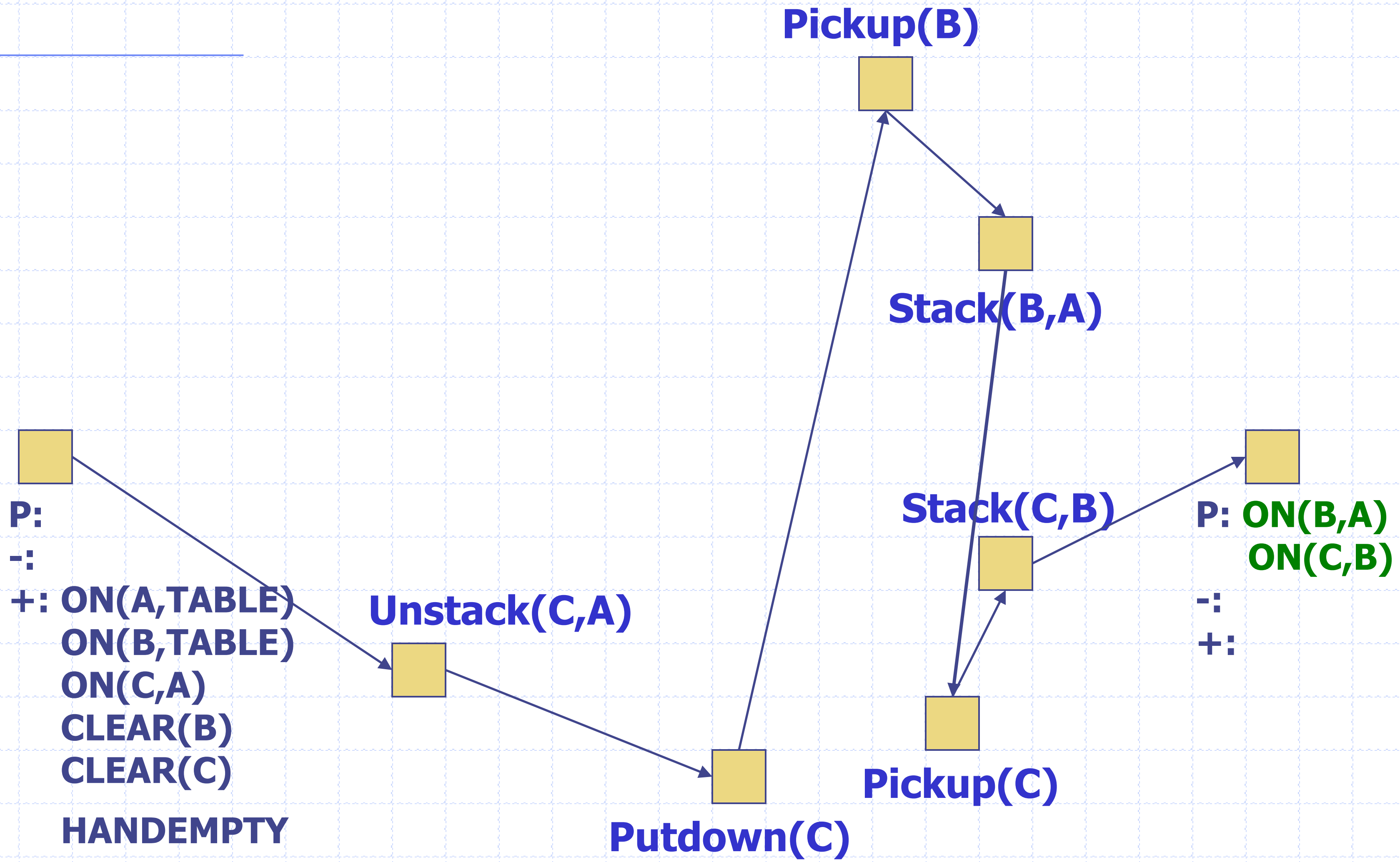
Putdown(C)

Pickup(C)

Stack(C,B)

P: ON(B,A)
ON(C,B)
-:
+:





P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)

HANDEMPTY

Unstack(C,A)

Putdown(C)

Pickup(B)

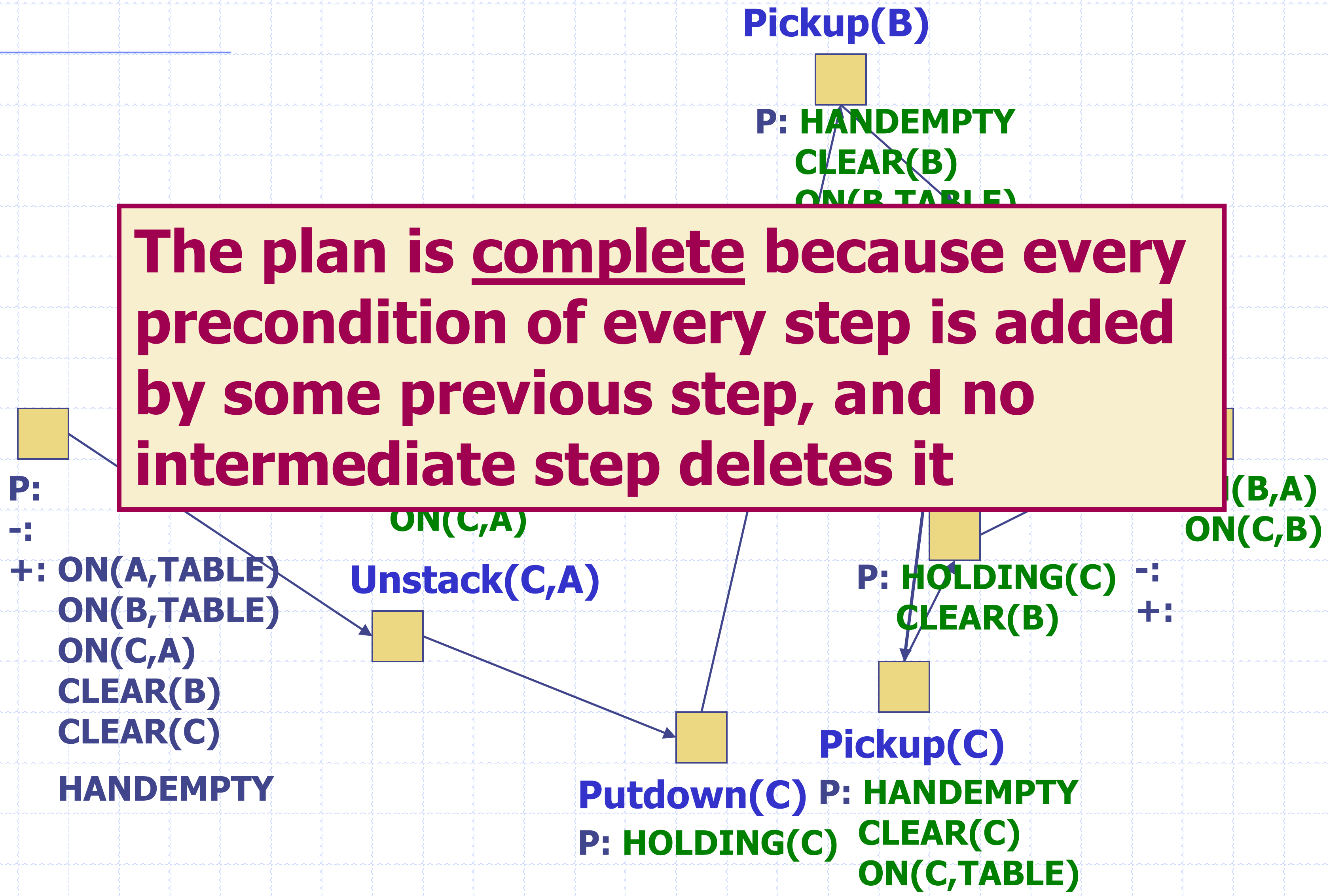
Stack(B,A)

Pickup(C)

Stack(C,B)

P: ON(B,A)
ON(C,B)
-:
+:

The plan is <u>complete</u> because every precondition of every step is added by some previous step, and no intermediate step deletes it



Pickup(B)

P: HANDEMPTY
CLEAR(B)
ON(B,TABLE)

P:
-:
+: ON(A,TABLE)
ON(B,TABLE)
ON(C,A)
CLEAR(B)
CLEAR(C)
HANDEMPTY

ON(C,A)

Unstack(C,A)

(B,A)
ON(C,B)

P: HOLDING(C)
CLEAR(B)

-:
+:

Pickup(C)
P: HANDEMPTY
CLEAR(C)
ON(C,TABLE)

Putdown(C)
P: HOLDING(C)

Planning Graph

- Is special data structure used for
  1. Deriving better heuristic estimates
  2. Extract a solution to the planning problem: GRAPHPLAN algorithm
- Is a sequence $\langle S_0, A_0, S_1, A_1, \ldots, S_i \rangle$ of levels
  - Alternating state levels & action levels
  - Levels correspond to time stamps
  - Starting at initial state
  - State level is a set of (propositional) literals
    - All those literals that could be true at that level
  - Action level is a set of (propositionalized) actions
    - All those actions whose preconditions appear in the state level (ignoring all negative interactions, etc.)
- Propositionalization may yield combinatorial explosition in the presence of a large number of objects

Example of a Planning Graph (1)

Init(Have(Cake))

Goal(Have(Cake)∧Eaten(Cake))

Action(Eat(Cake)

Precond: Have(Cake)

Effect: ¬Have(Cake)∧Eaten(Cake))

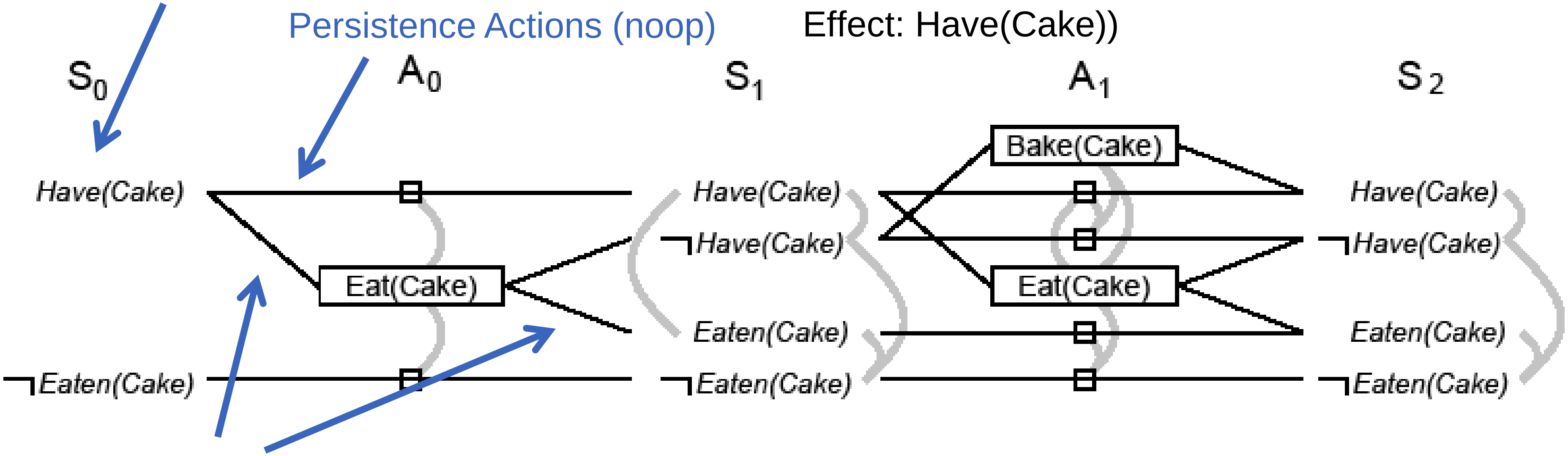
Action(Bake(Cake)

Precond: ¬Have(Cake)

Effect: Have(Cake))

Propositions true at the initial state

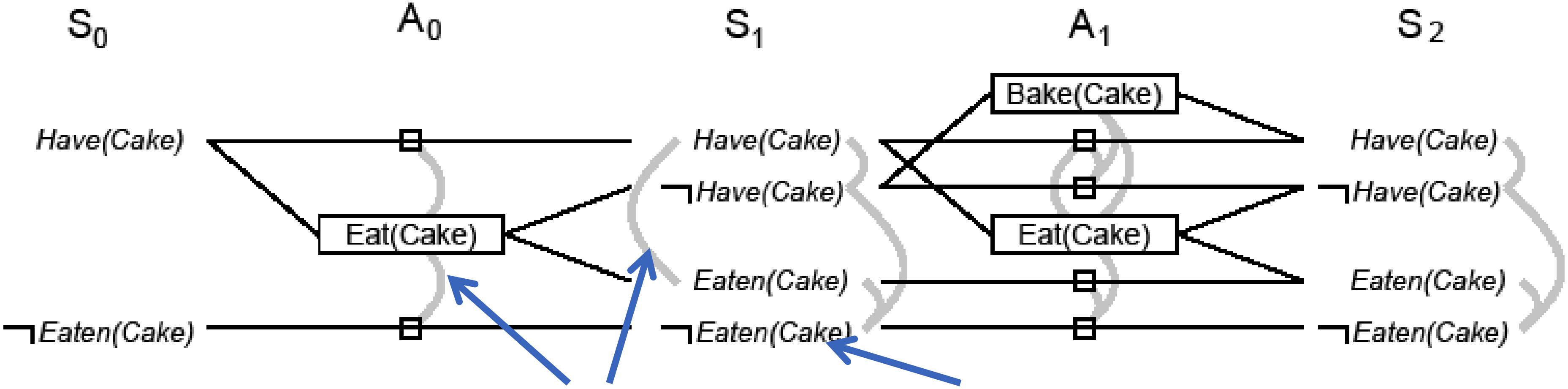
Persistence Actions (noop)



Action is connected to its preconds & effects

Example of a Planning Graph (2)

- At each state level, list all literals that may hold at that level
- At each action level, list all noops & all actions whose preconditions may hold at previous levels
- If some goal literal does not appear in the final level of the graph, the goal is not achievable
  - Repeat until plan ‘levels contains all the literal of the goal
  - Terminate if Si = Si+1
- Building the Planning Graph is a polynomial process
- Add (binary) mutual exclusion (mutex) links between conflicting actions and between conflicting literals

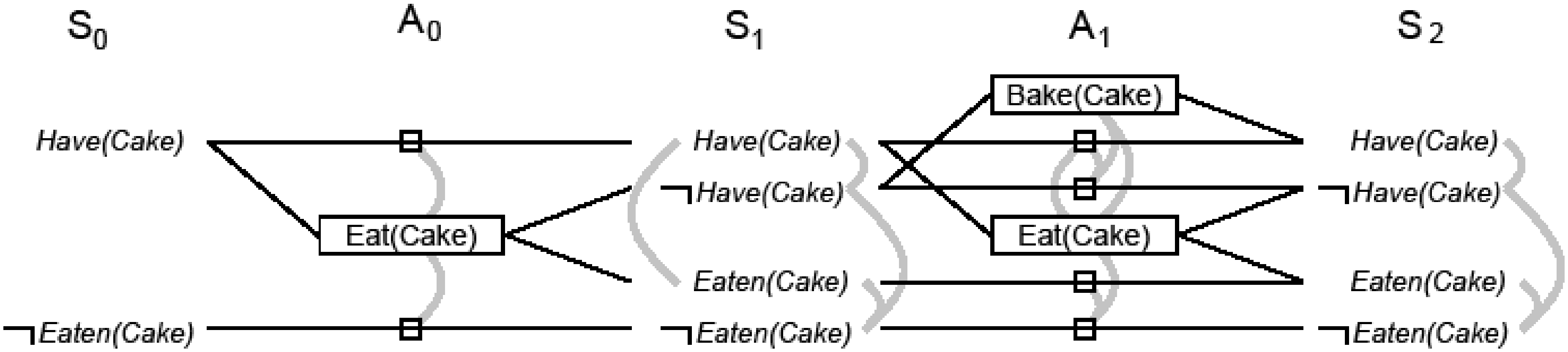


Mutual exclusion links

S_1 represents multiple states

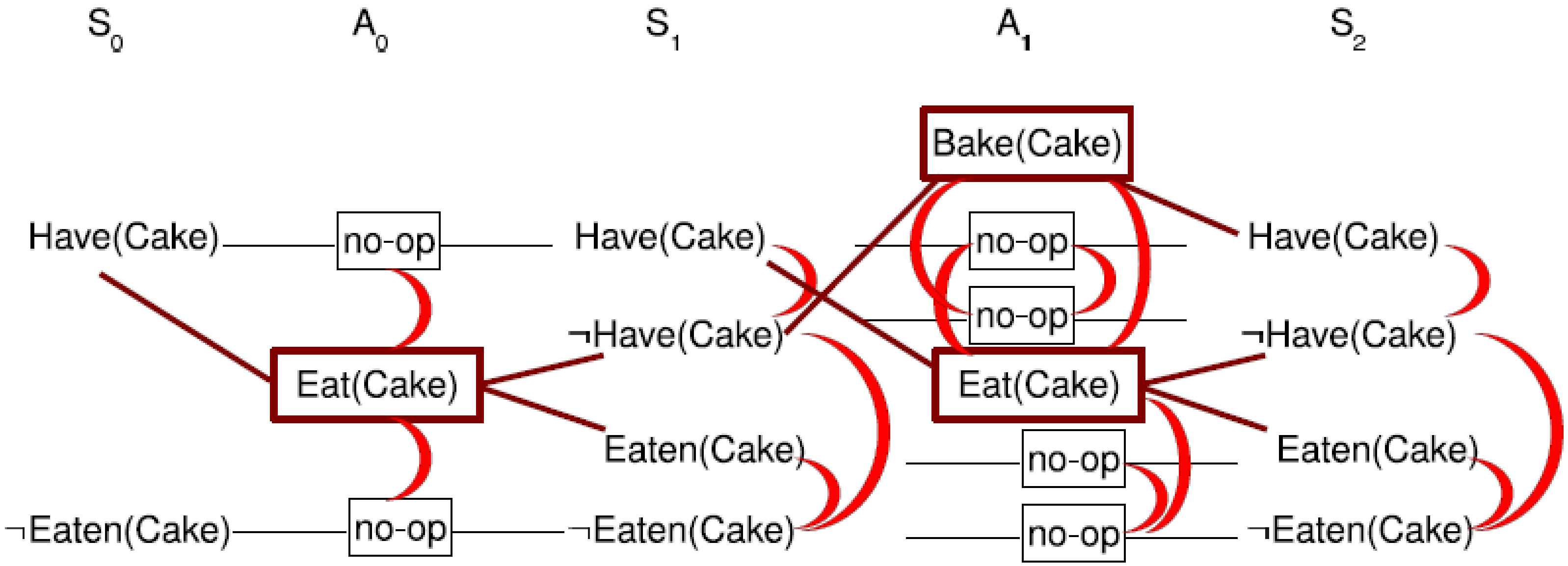
Mutex Links between Actions

1. **Inconsistent effects**: one action negates an effect of another
   - Eat(Cake) & noop of Have(Cake) disagree on effect Have(Cake)
2. **Interference**: An action effect negates the precondition of another
   - Eat(Cake) negates precondition of the noop of Have(Cake):
3. **Competing needs**: A precondition on an action is mutex with the precondition of another
   - Bake(Cake) & Eat(Cake): compete on Have(Cake) precondition

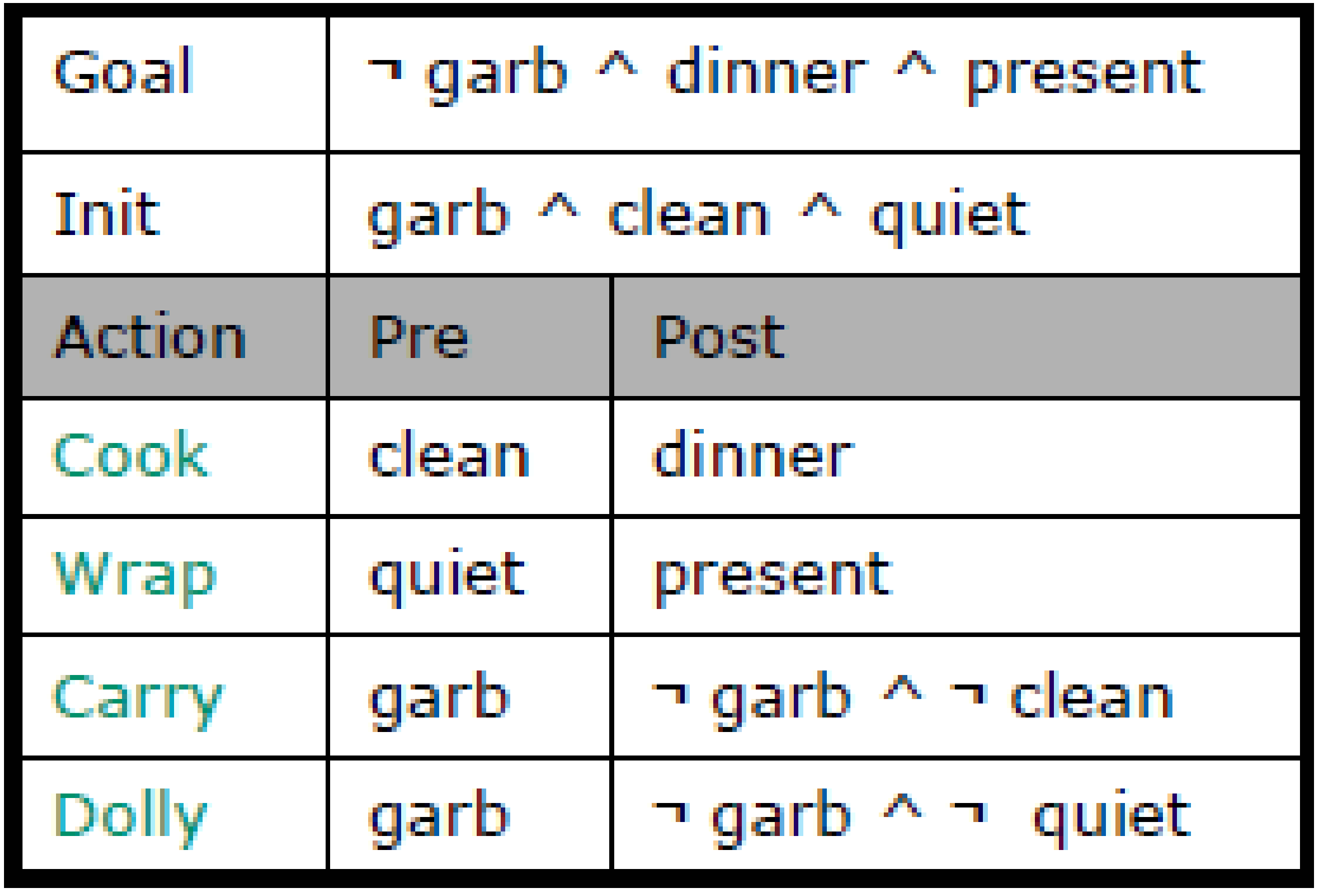


Mutex Links between Literals

1. Two literals are negation of each other
2. **Inconsistent support**: Each pair of actions that can achieve the two literals is mutex. Examples:
   - In S1, Have(Cake) & Eaten(Cake) are mutex
   - In S2, they are not because Bake(Cake) & the noop of Eaten(Cake) are not mutex

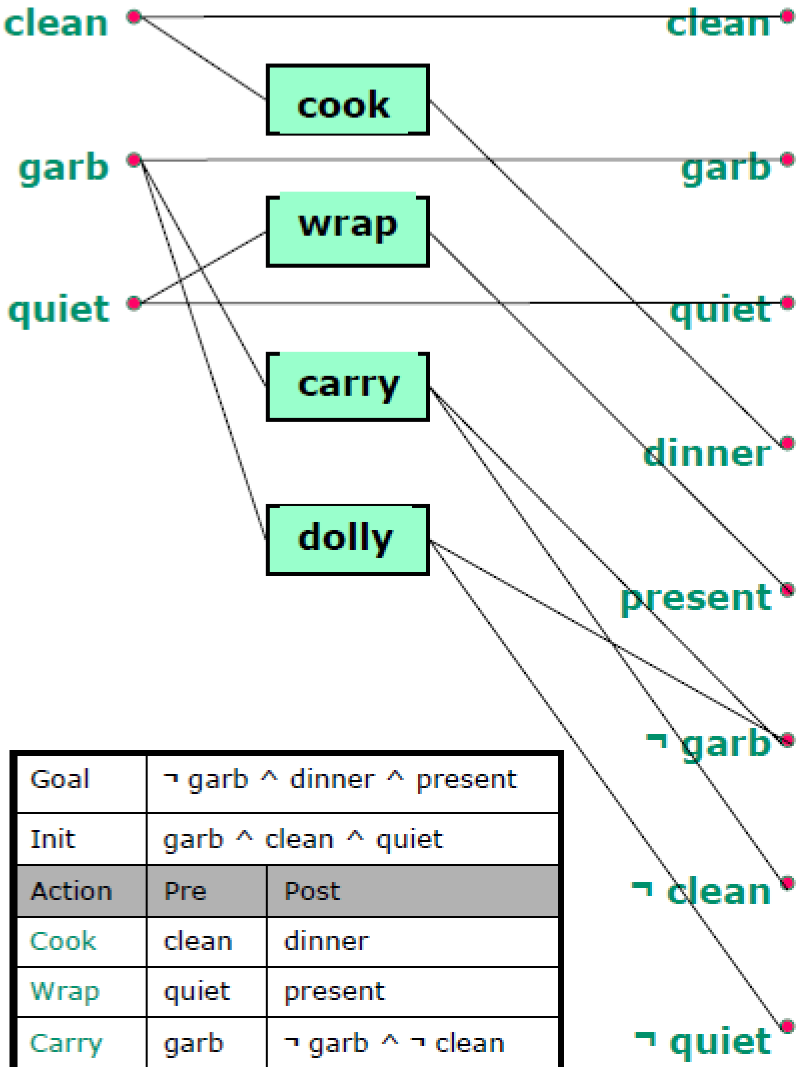


Birthday Dinner Example



Goal	¬ garb ^ dinner ^ present	
Init	garb ^ clean ^ quiet	
Action	**Pre**	**Post**
Cook	clean	dinner
Wrap	quiet	present
Carry	garb	¬ garb ^ ¬ clean
Dolly	garb	¬ garb ^ ¬ quiet

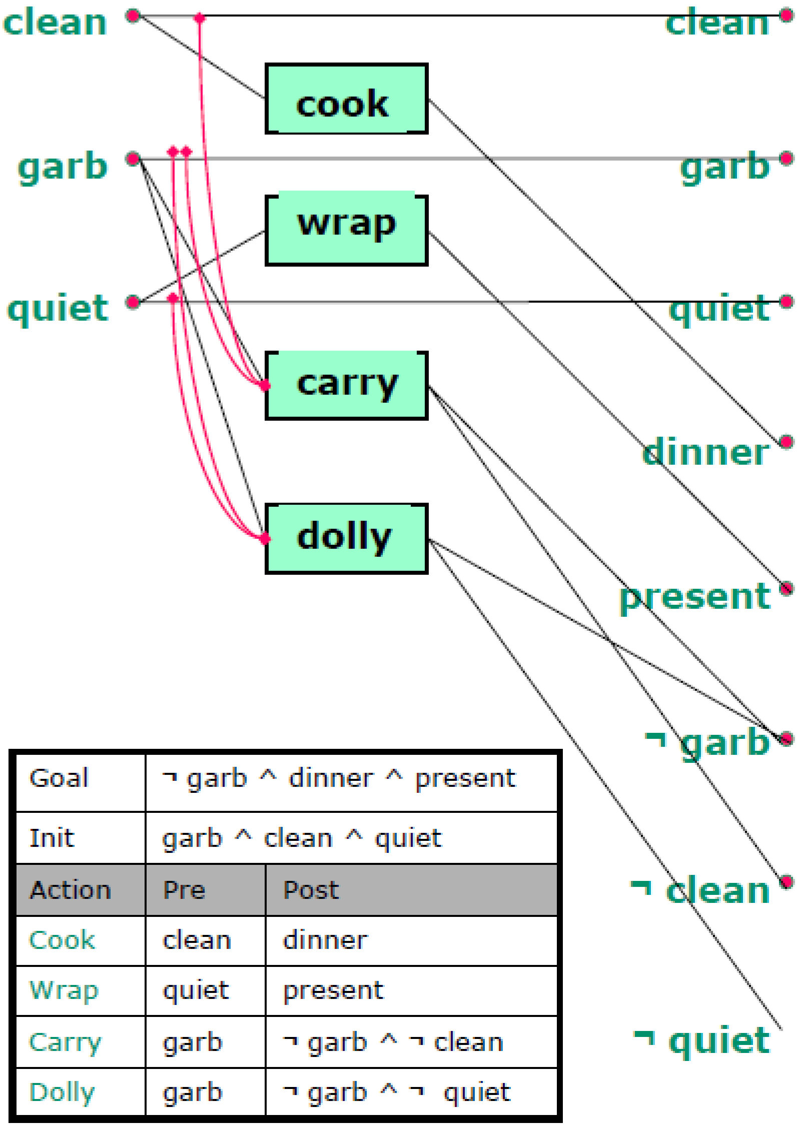
Planning Graph



Goal	¬ garb ^ dinner ^ present	
Init	garb ^ clean ^ quiet	
Action	Pre	Post
Cook	clean	dinner
Wrap	quiet	present
Carry	garb	¬ garb ^ ¬ clean
Dolly	garb	¬ garb ^ ¬ quiet

Mutex between Actions

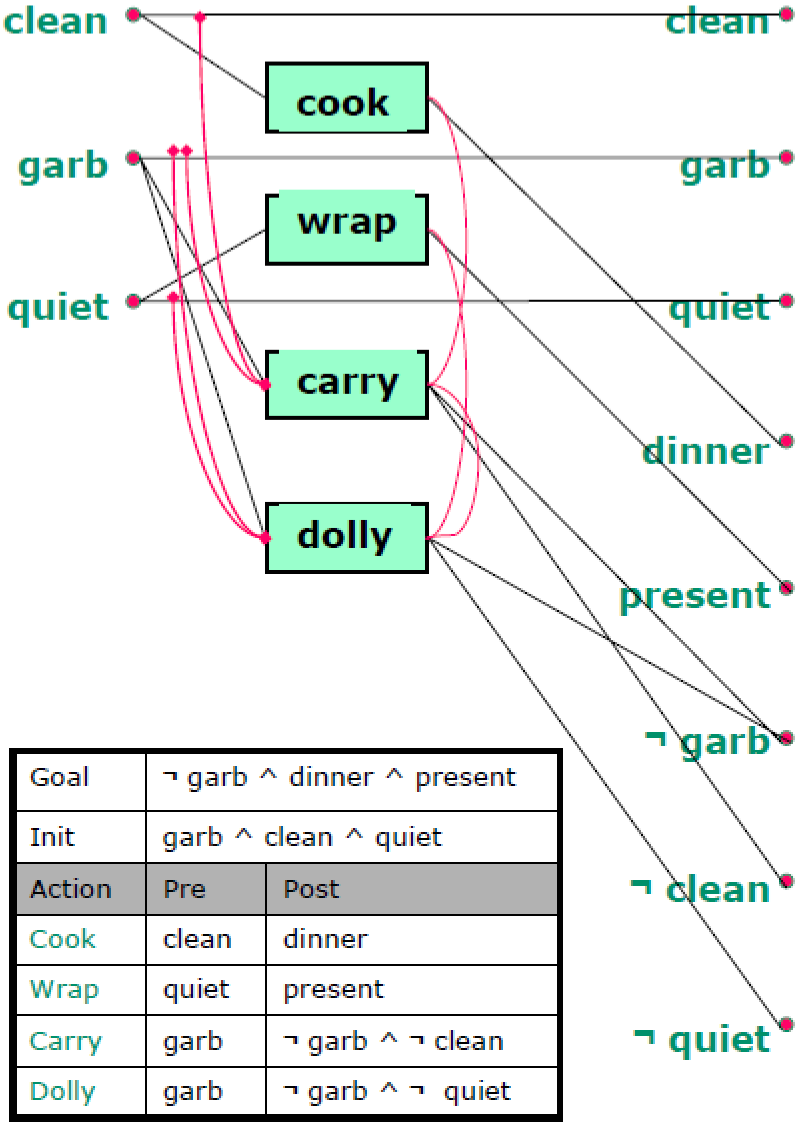
- The first reason that actions can be mutex is due to inconsistent effects.
  - Carry and maintaining clean have inconsistent effects (because carry makes clean false).
  - Maintaining garb has inconsistent effects with both carry and dolly (which make garb false).
  - Maintaining quiet has inconsistent effects with dolly (which makes quiet false).



Goal	¬ garb ^ dinner ^ present	
Init	garb ^ clean ^ quiet	
Action	Pre	Post
Cook	clean	dinner
Wrap	quiet	present
Carry	garb	¬ garb ^ ¬ clean
Dolly	garb	¬ garb ^ ¬ quiet

Mutex between Actions

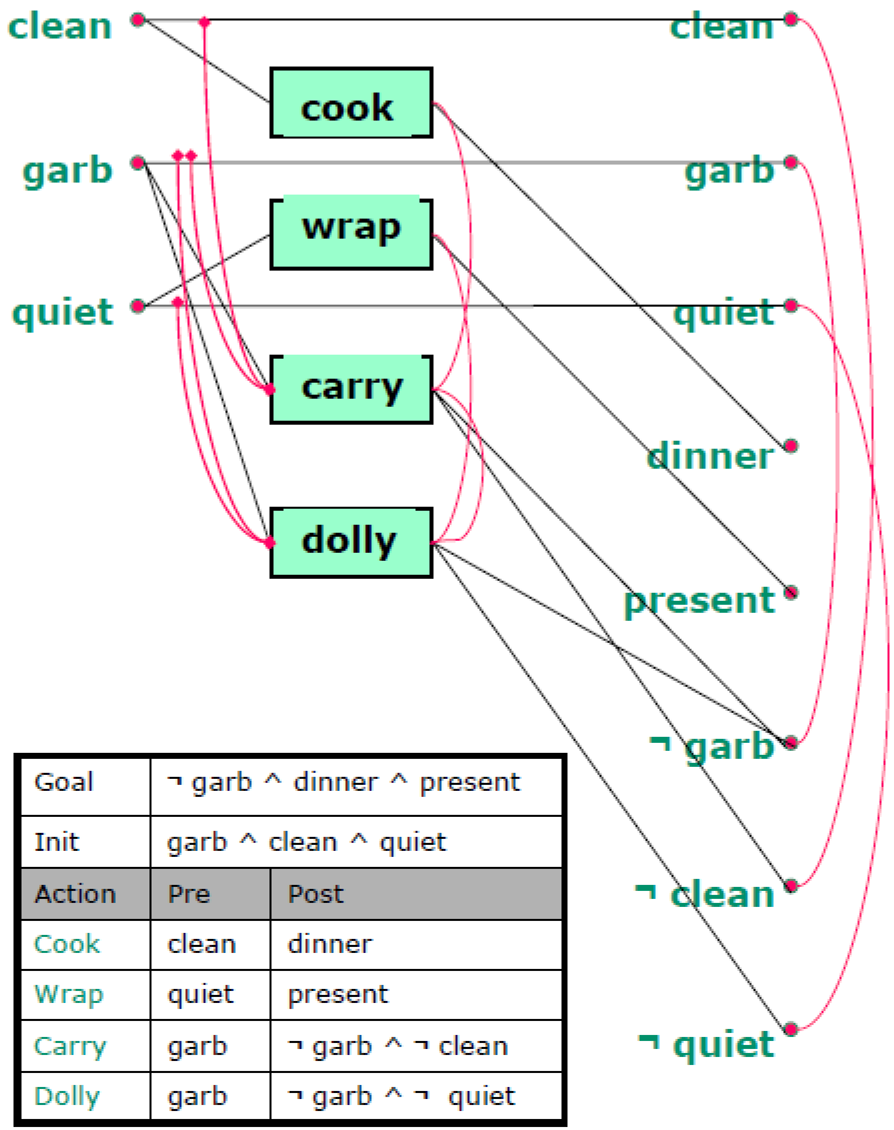
- Another kind of mutex is due to interference: one action negates the precondition of another.
    - We have interference between cook and carry (carry makes clean false, which is required for cook)
    - also have interference between wrap and dolly (dolly makes quiet false, which is required for wrap.).
    - we have interference between carry and dolly, because they each require that garbage be present, and they each remove it.



Goal	¬ garb ^ dinner ^ present	
Init	garb ^ clean ^ quiet	
Action	Pre	Post
Cook	clean	dinner
Wrap	quiet	present
Carry	garb	¬ garb ^ ¬ clean
Dolly	garb	¬ garb ^ ¬ quiet

Mutex Links between Literals

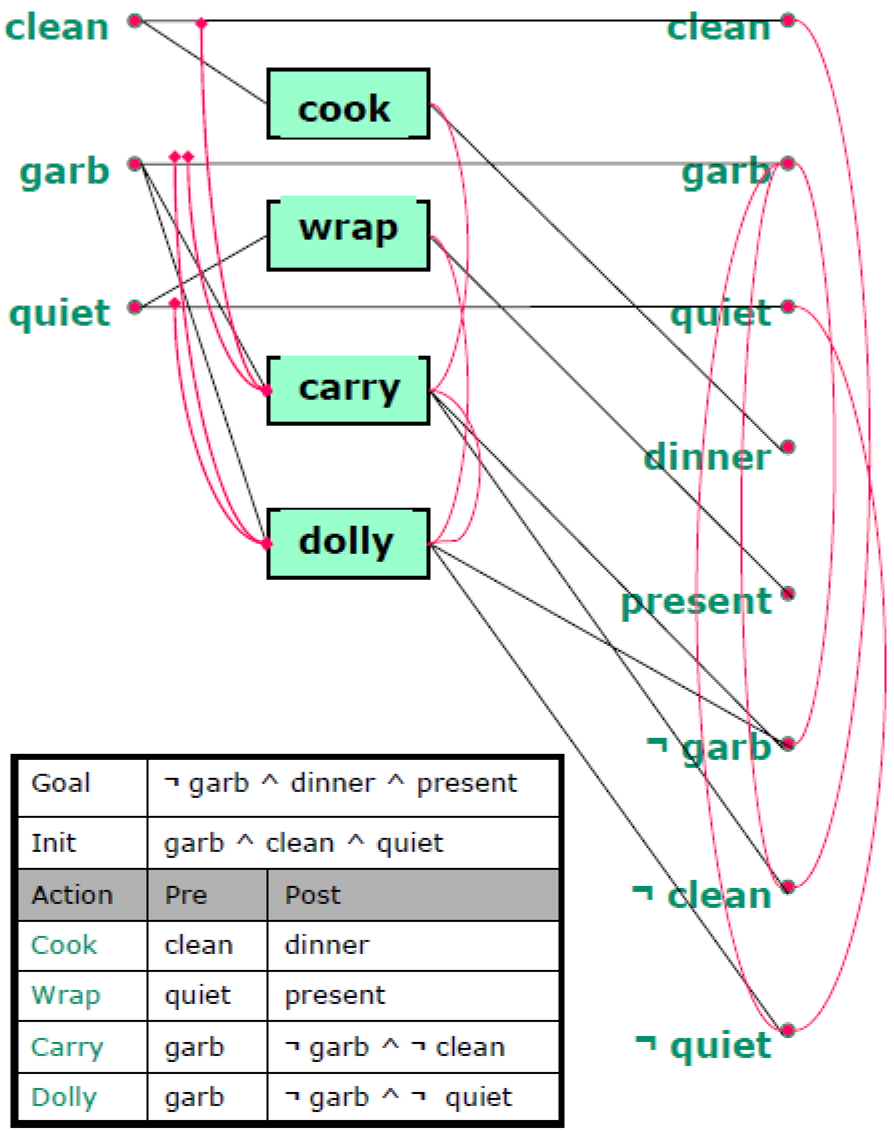
- First of all, every proposition is mutex with its negation.



Goal	¬ garb ^ dinner ^ present	
Init	garb ^ clean ^ quiet	
Action	Pre	Post
Cook	clean	dinner
Wrap	quiet	present
Carry	garb	¬ garb ^ ¬ clean
Dolly	garb	¬ garb ^ ¬ quiet

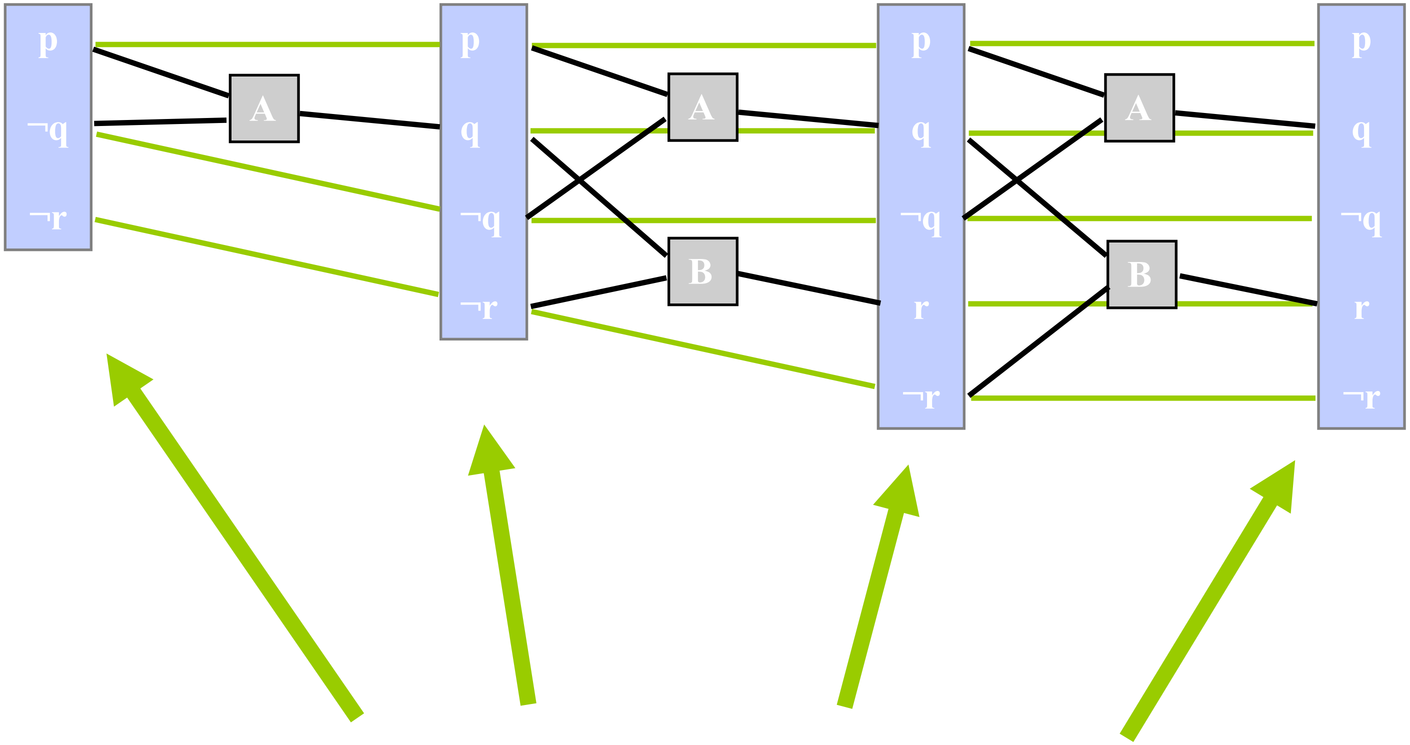
Mutex Links between Literals

- The other reason we might have mutexes is because of inconsistent support (all ways of achieving the propositions are pairwise mutex).
- Here we have that garbage is mutex with not clean and with not quiet (the only way to make garbage true is to maintain it, which is mutex with carry and with dolly).
- Dinner is mutex with not clean because cook and carry, the only way of achieving these propositions, are mutex at the previous level.
- And present is mutex with not quiet because wrap and dolly are mutex at the previous level
- Finally not clean is mutex with not quiet because carry and dolly are mutex at the previous level.



Goal	¬ garb ^ dinner ^ present	
Init	garb ^ clean ^ quiet	
Action	Pre	Post
Cook	clean	dinner
Wrap	quiet	present
Carry	garb	¬ garb ^ ¬ clean
Dolly	garb	¬ garb ^ ¬ quiet

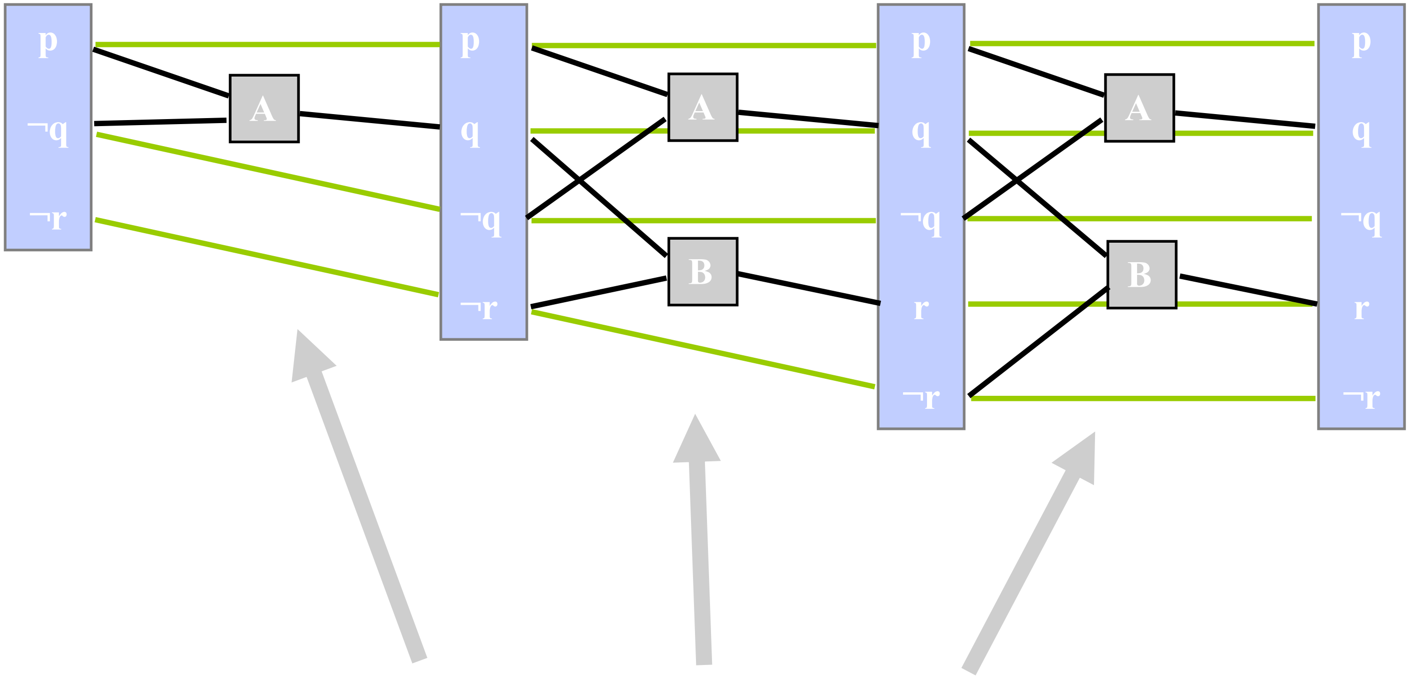
Planning Graph Propery 1



Propositions monotonically increase

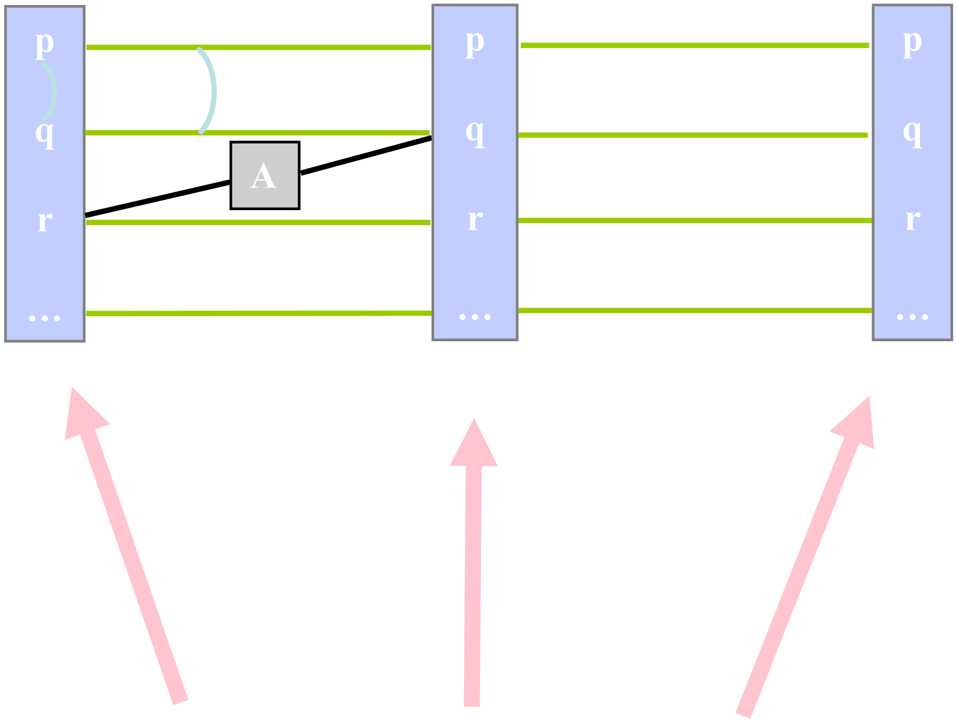
(always carried forward by no-ops)

Planning Graph Propery 2



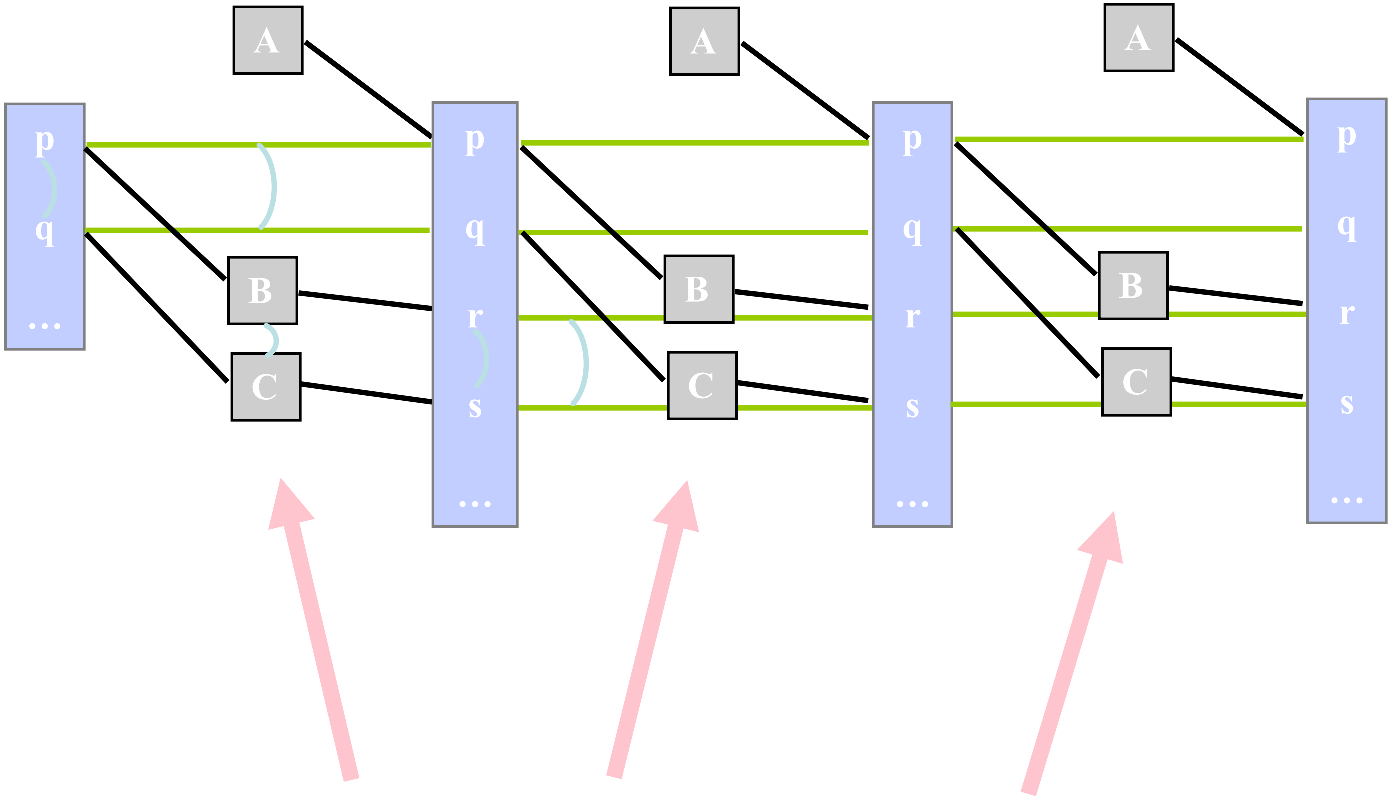
Actions monotonically increase

Planning Graph Properties 3



- Proposition mutex relationships monotonically decrease
- Specifically, if p and q are in layer n and are not mutex then they will not be mutex in future layers.

Planning Graph Propery 4



Action mutex relationships monotonically decrease

Properties 5

Planning Graph ‘levels off’.

- After some time k all levels are identical
  - In terms of propositions, actions, mutexes
- This is because there are a finite number of propositions and actions, the set of literals never decreases and mutexes don’t reappear.

Planning Graph Important Ideas

- Plan graph construction is polynomial time
  - Though construction can be expensive when there are many "objects" and hence many propositions

- The plan graph captures important properties of the planning problem
  - Necessarily unreachable literals and actions
  - Possibly reachable literals and actions
  - Mutually exclusive literals and actions

- Significantly prunes search space compared to previously considered planners

- Plan graphs can also be used for deriving admissible (and good non-admissible) heuristics

Planning Graph for Heuristic Estimation

- A literal that does not appear in the final level cannot be achieved by any plan
  - State-space search: Any state containing an unachievable literal has cost $h(n)=\propto$
  - POP: Any plan with an unachievable open condition has cost $h(n)=\propto$
- The cost of achieving any goal literal can be estimated by counting the number of levels before it appears
  - This heuristic never overestimates
  - Estimate can be improved by serializing the graph (serial planning graph: one action per level) by adding mutex between all actions in a given level
- The estimate of a conjunction of goal literals
  - Three heuristics: max level, level sum, set level

Estimate of Conjunction of Goal Literals

- Max-level
  - The largest level of a literal in the conjunction
  - Admissible, not very accurate
- Level sum heuristic,
  - following the subgoal independence assumption, returns the sum of the level costs of the goals; this is inadmissible but works very well in practice for problems that are largely decomposable
- Set level
  - Finds the level at which all literals appear without any pair of them being mutex
  - Dominates max-level, works extremely well on problems where there is a great deal of interaction among subplans

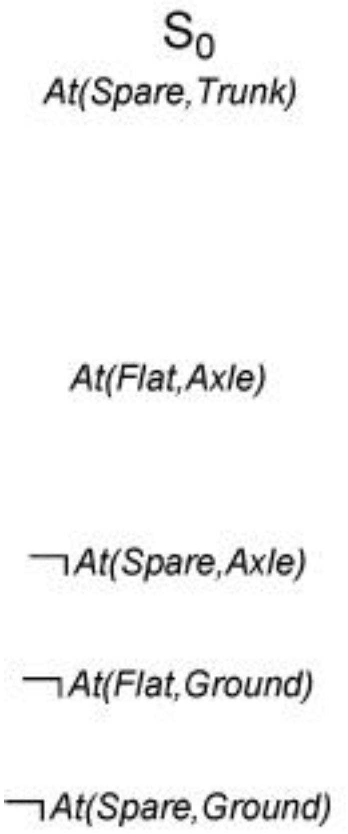
GraphPlan algorithm

```
GraphPlan(problem) returns solution or failure
graph ← InitialPlanningGraph(problem)
goals ← Goals[problem]
loop do
    if goals all non-mutex in last level of graph then do
        solution ← ExtractSolution(graph,goals,Length(graph))
        if solution ≠ failure then return solution
        else if  NoSolutionPossible(graph) then return failure
    graph ← ExpandGraph (graph,problem)
```

- Two main stages
    1. Extract solution
    2. Expand the graph

Example of GRAPHPLAN Execution (1)

- At(Spare,Axle) is not in S_0
- No need to extract solution
- Expand the plan



S_0

At(Spare,Trunk)

At(Flat,Axle)

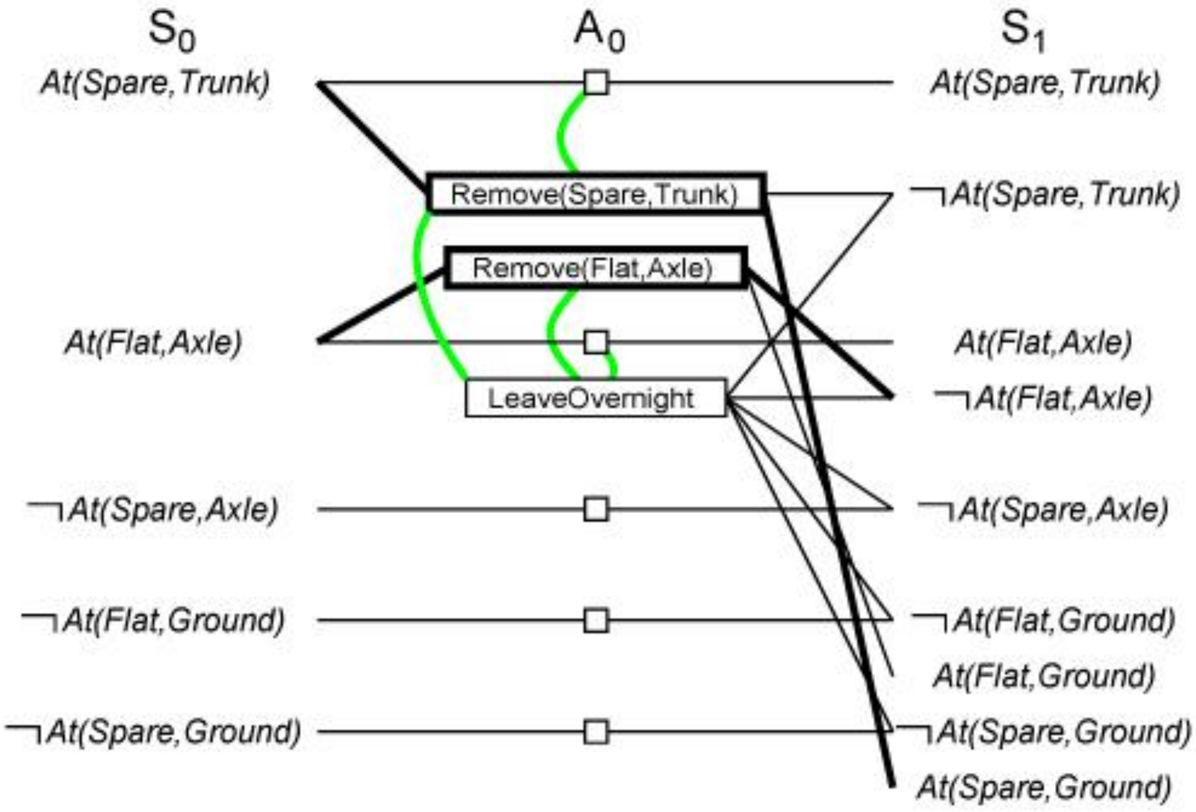
¬At(Spare,Axle)

¬At(Flat,Ground)

¬At(Spare,Ground)

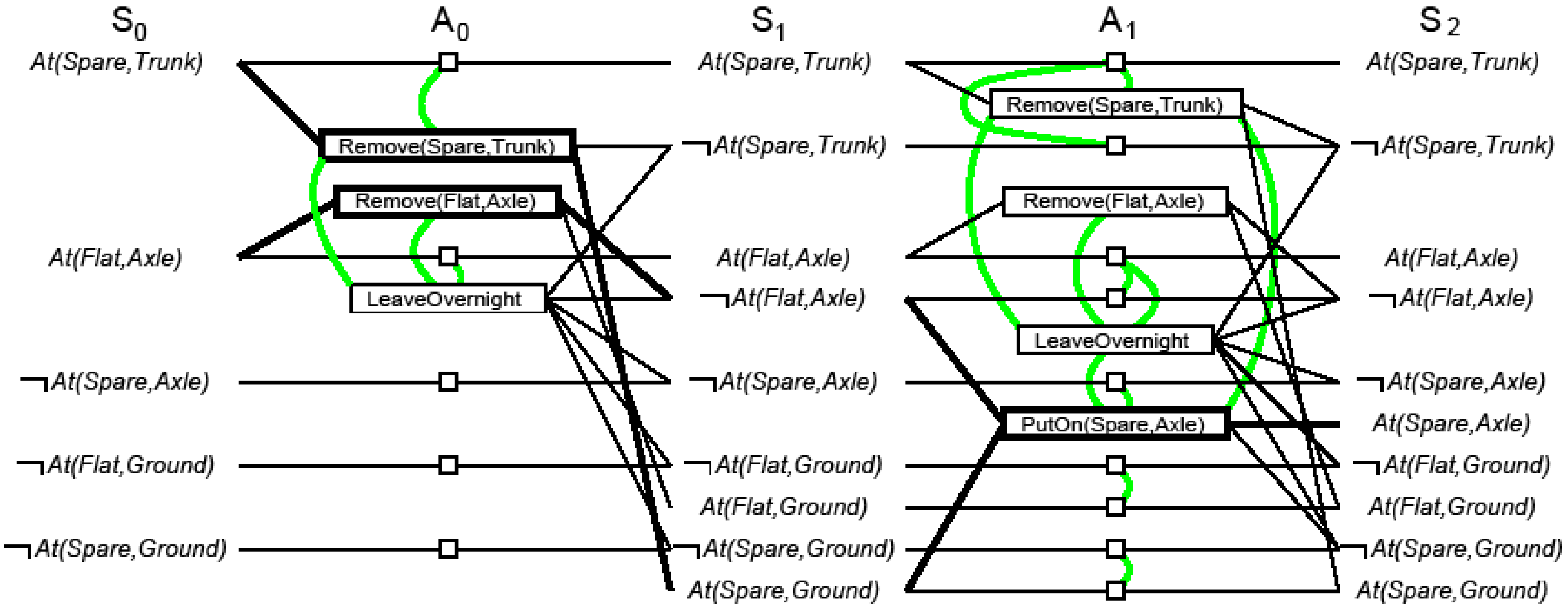
Example of GRAPHPLAN Execution (2)

- Three actions are applicable
- 3 actions and 5 noops are added
- Mutex links are added
- At(Spare,Axle) still not in S_1
- Plan is expanded



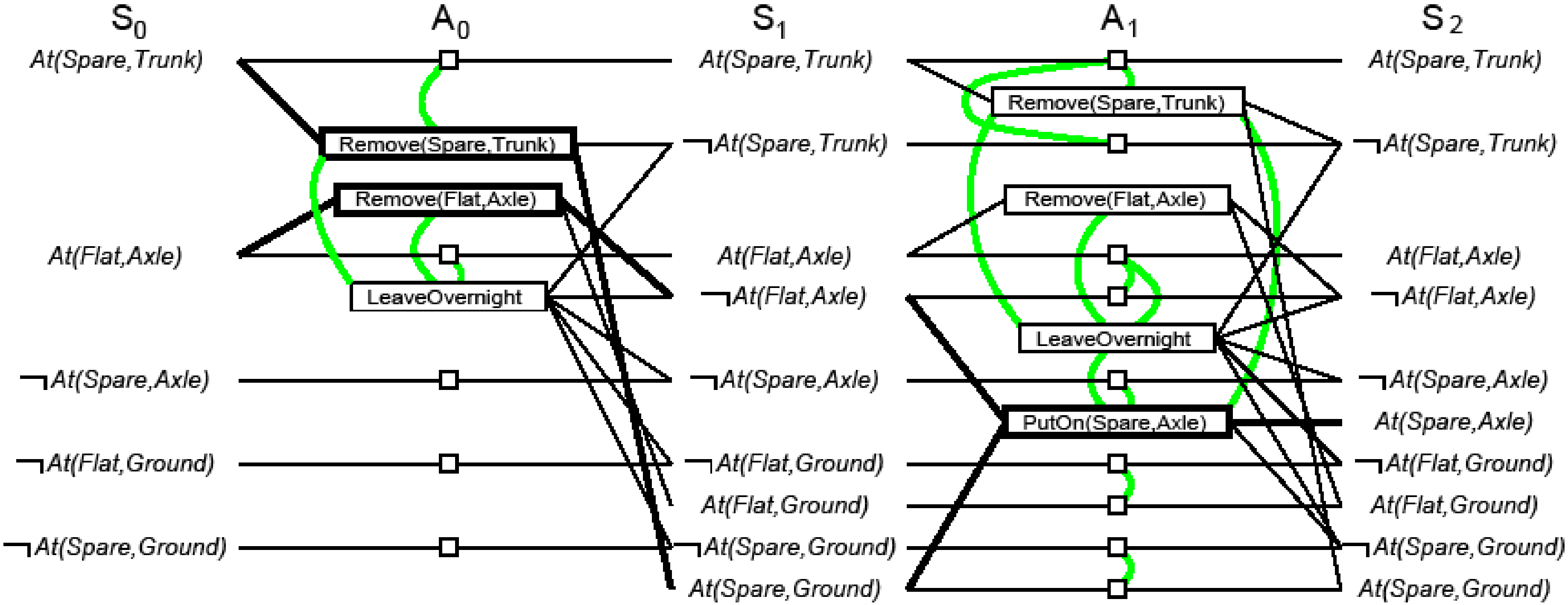
Example of GRAPHPLAN Execution (3)

- Illustrates well mutex links: inconsistent effects, interference, competing needs, inconsistent support



Solution Extraction (Backward)

1. Solve a Boolean CSP: Variables are actions, domains are {0=out of plan, 1=in plan), constraints are mutex
2. Search problem from last level backward



Backtrack Search for Solution Extraction

- Starting at the highest fact level
  - Each goal is put in a goal list for the current fact layer
  - Search iterates thru each fact in the goal list trying to find an action to support it which is not mutex with any other chosen action
  - When an action is chosen, its preconditions are added to the goal list of the lower level
  - When all facts in the goal list of the current level have a consistent assignment of actions, the search moves to the next level

- Search backtracks to the previous level when it fails to assign an action to each fact in the goal list at a given level

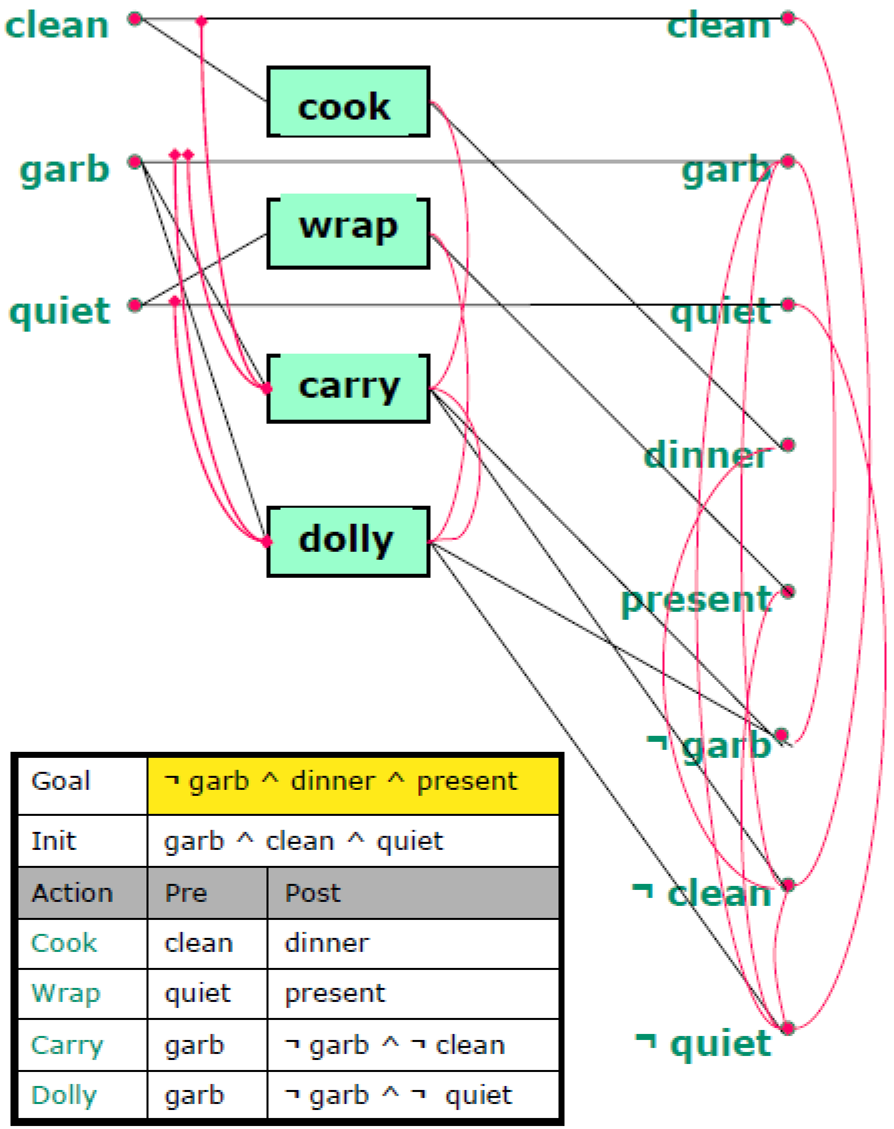
- Search succeeds when the first level is reached.

Example of GRAPHPLAN Execution

- For this particular problem, we start at S2 with the goal At(Spare, Axle).
- The only choice we have for achieving the goal set is PutOn(Spare, Axle).
- That brings us to a search state at S1 with goals At(Spare, Ground) and ¬At(Flat, Axle).
- The former can be achieved only by Remove(Spare, Trunk), and the latter by either Remove(Flat, Axle) or LeaveOvernight.
- But LeaveOvernight is mutex with Remove(Spare, Trunk), so the only solution is to choose Remove(Spare, Trunk) and Remove(Flat, Axle).
- That brings us to a search state at S0 with the goals At(Spare, Trunk) and At(Flat, Axle).
- Both of these are present in the state, so we have a solution: the actions Remove(Spare, Trunk) and Remove(Flat, Axle) in level A0, followed by PutOn(Spare, Axle) in A1.

Birthday Dinner Example

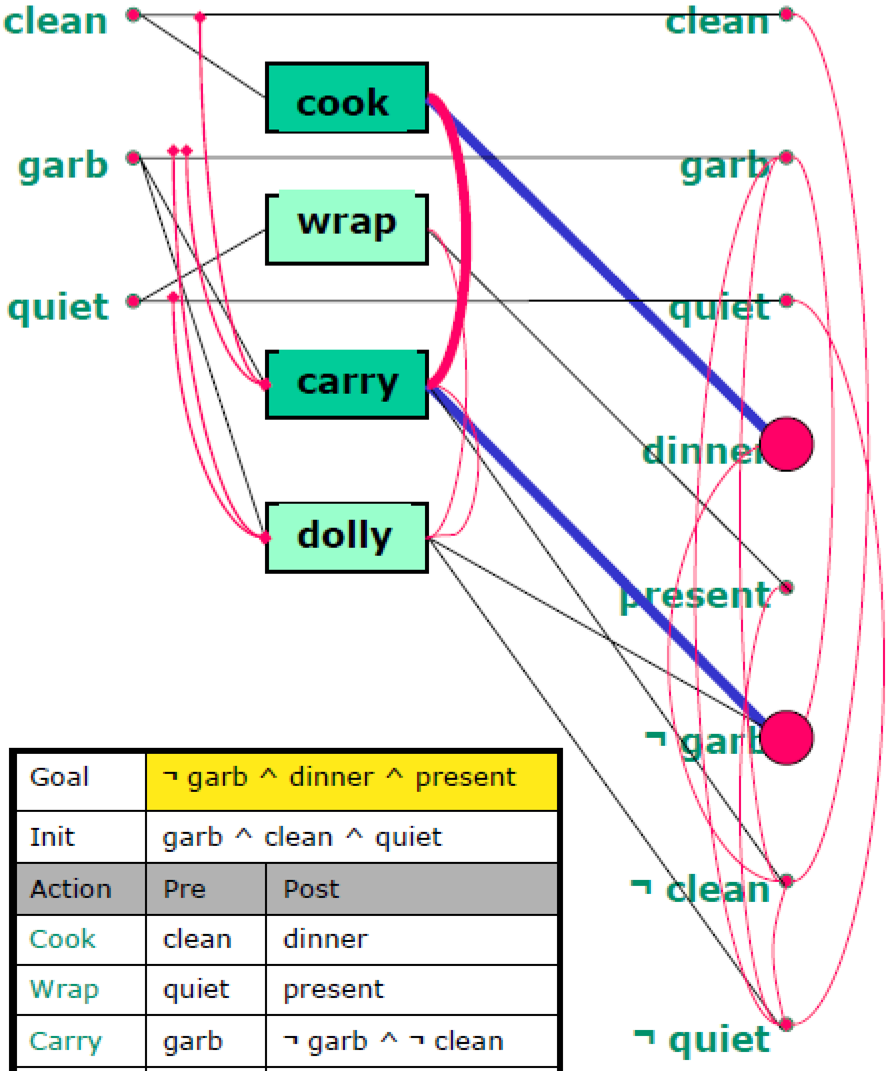
- Our goal is **!garbage and dinner and present.**
- Layer 2 contains !garbage and dinner and present.
- So it looks like these could possibly be true. They're not obviously inconsistent.



Goal	¬ garb ^ dinner ^ present	
Init	garb ^ clean ^ quiet	
Action	Pre	Post
Cook	clean	dinner
Wrap	quiet	present
Carry	garb	¬ garb ^ ¬ clean
Dolly	garb	¬ garb ^ ¬ quiet

Birthday Dinner Example

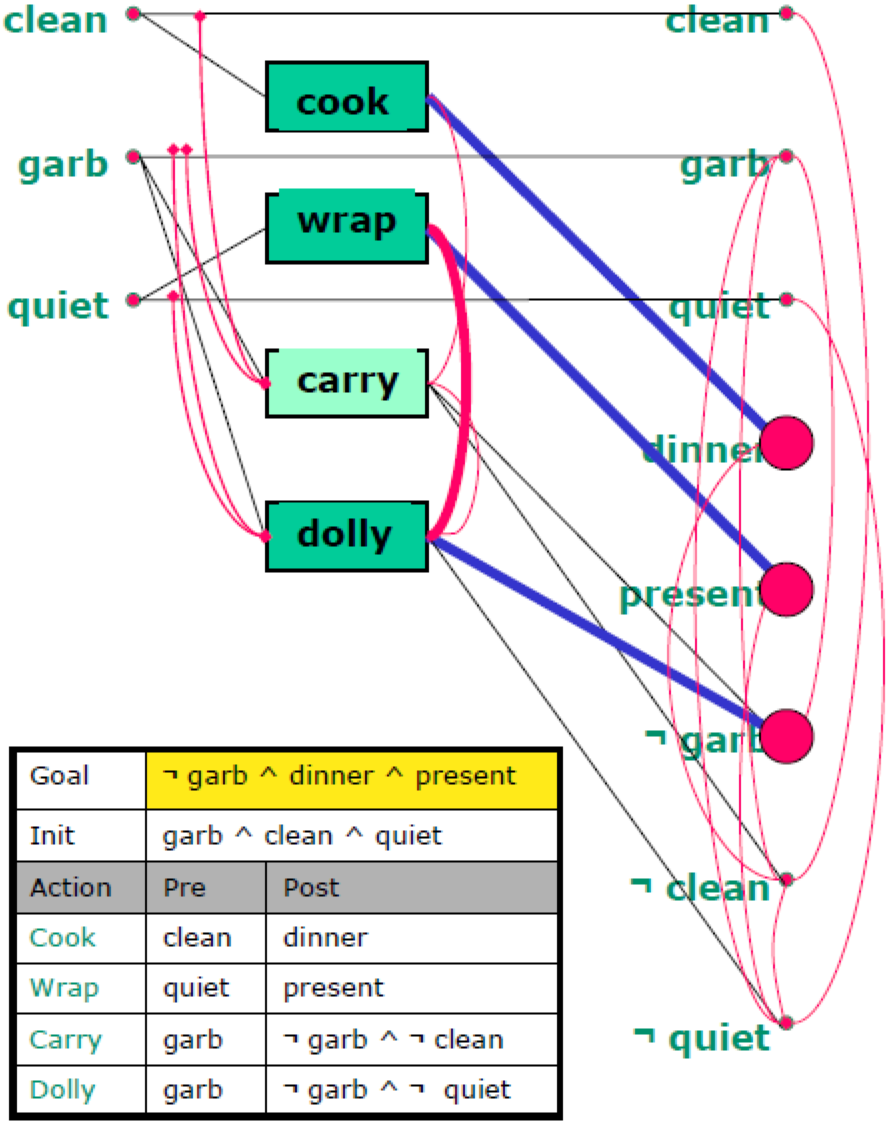
- So, we'll start looking for a plan by finding a way to make not garbage true.
- We'll try using the carry action.
- Now, we'll try to make dinner true the only way we can, with the cook action.
- But cook and carry are mutex, so this won't work.



Goal	¬ garb ^ dinner ^ present	
Init	garb ^ clean ^ quiet	
Action	Pre	Post
Cook	clean	dinner
Wrap	quiet	present
Carry	garb	¬ garb ^ ¬ clean
Dolly	garb	¬ garb ^ ¬ quiet

Birthday Dinner Example

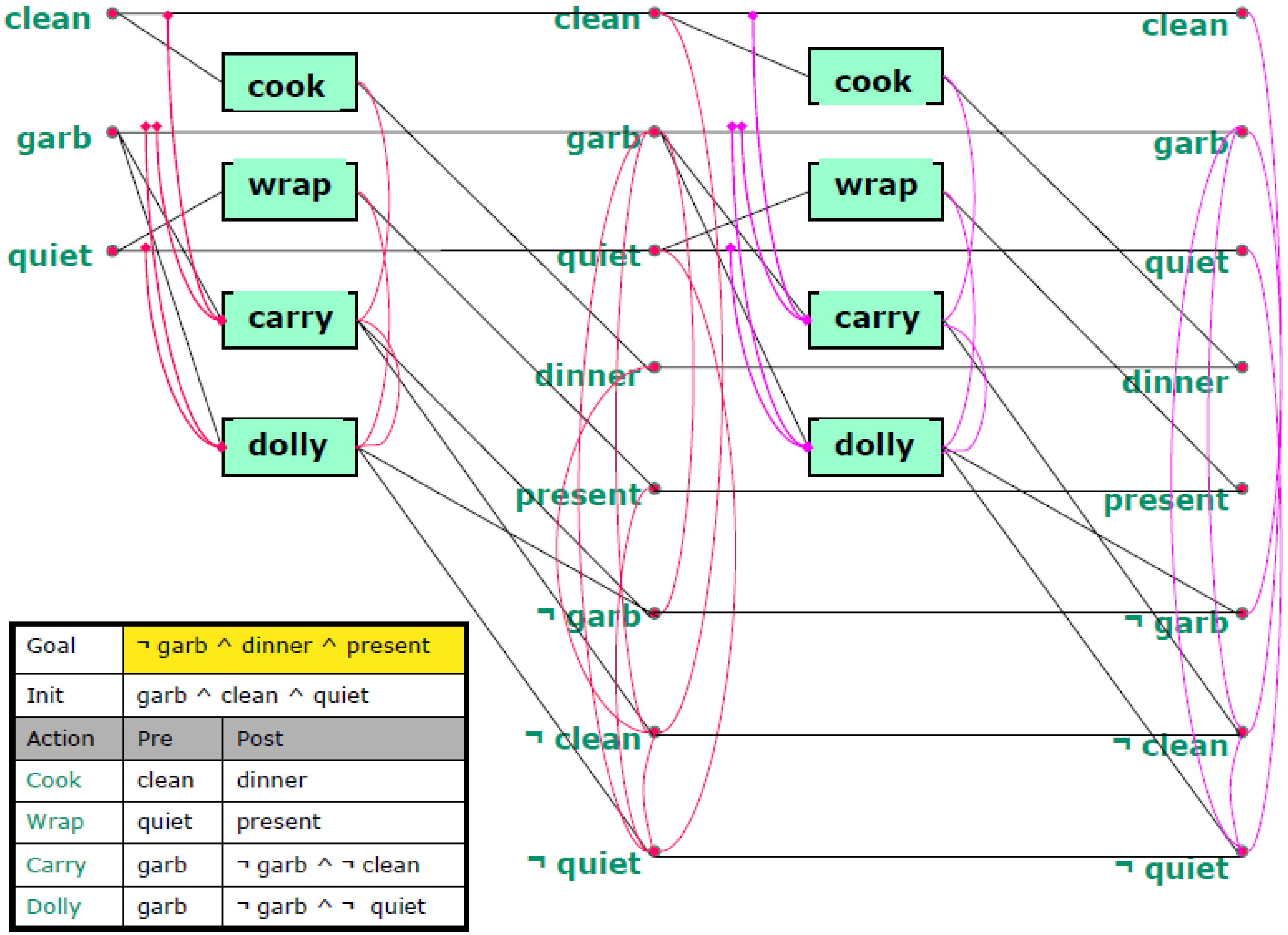
- Because there aren’t any other ways to make dinner, we fail, and have to try a different way of making not garbage true. This time, we’ll try dolly.
- Now, we can cook dinner, and we don’t have any mutex problems with dolly.
- We have to make present true as well. The only way of doing that is with wrap, but wrap is mutex with dolly. So, we fail completely.



Goal	¬ garb ^ dinner ^ present	
Init	garb ^ clean ^ quiet	
Action	Pre	Post
Cook	clean	dinner
Wrap	quiet	present
Carry	garb	¬ garb ^ ¬ clean
Dolly	garb	¬ garb ^ ¬ quiet

Birthday Dinner Example

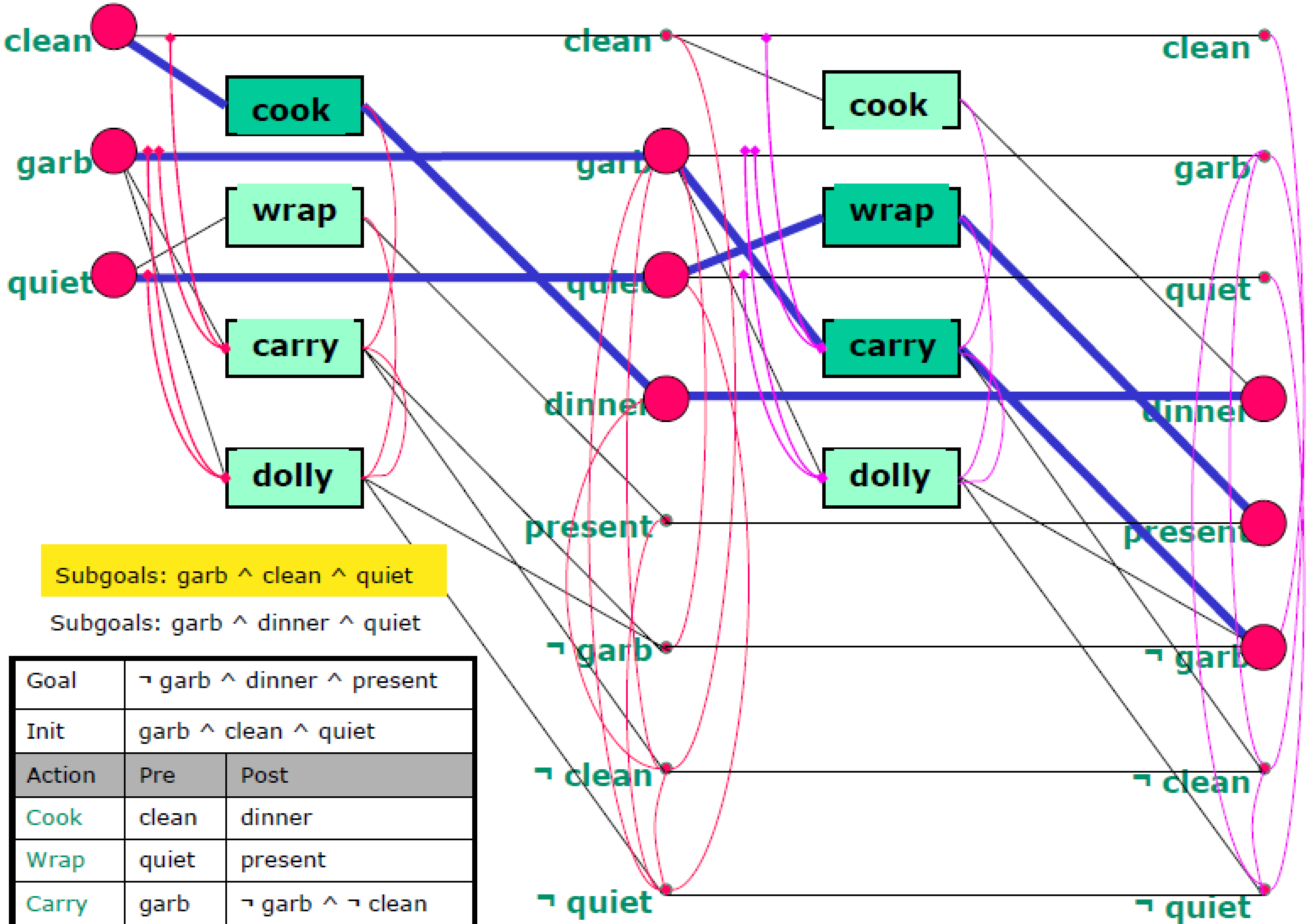
- There's no way to achieve all of these goals in parallel. So we have to consider a depth two plan.
- We start by adding another layer to the plan graph
- Then find and show the Mutexs on the next level



Goal	¬ garb ^ dinner ^ present	
Init	garb ^ clean ^ quiet	
Action	Pre	Post
Cook	clean	dinner
Wrap	quiet	present
Carry	garb	¬ garb ^ ¬ clean
Dolly	garb	¬ garb ^ ¬ quiet

Birthday Dinner Example

- Final Solution



Subgoals: garb ^ clean ^ quiet

Subgoals: garb ^ dinner ^ quiet

Goal	¬ garb ^ dinner ^ present	
Init	garb ^ clean ^ quiet	
Action	Pre	Post
Cook	clean	dinner
Wrap	quiet	present
Carry	garb	¬ garb ^ ¬ clean
Dolly	garb	¬ garb ^ ¬ quiet

Improving search using Heuristic

- The search may still degenerate to an exponential exploration
- **Heuristic:**
  1. Pick the literal with a highest level cost
  2. To achieve this literal, pick actions with easiest preconditions (the set of preconditions which has the smallest max level cost)

Termination of GRAPHPLAN

- GRAPHPLAN is guaranteed to terminate
  - Literal increase monotonically
  - Actions increase monotonically
  - Mutexes decrease monotinically
- A solution is guaranteed not to exist when
  - The graph levels off with all goals present & non-mutex, and
  - EXTRACTSOLUTION fails to find solution

Optimality of GRAPHPLAN

- The plans generated by GRAPHPLAN
  - Are optimal in the number of steps needed to execute the plan
  - Not necessarily optimal in the number of actions in the plan (GRAPHPLAN produces partially ordered plans)