Planning

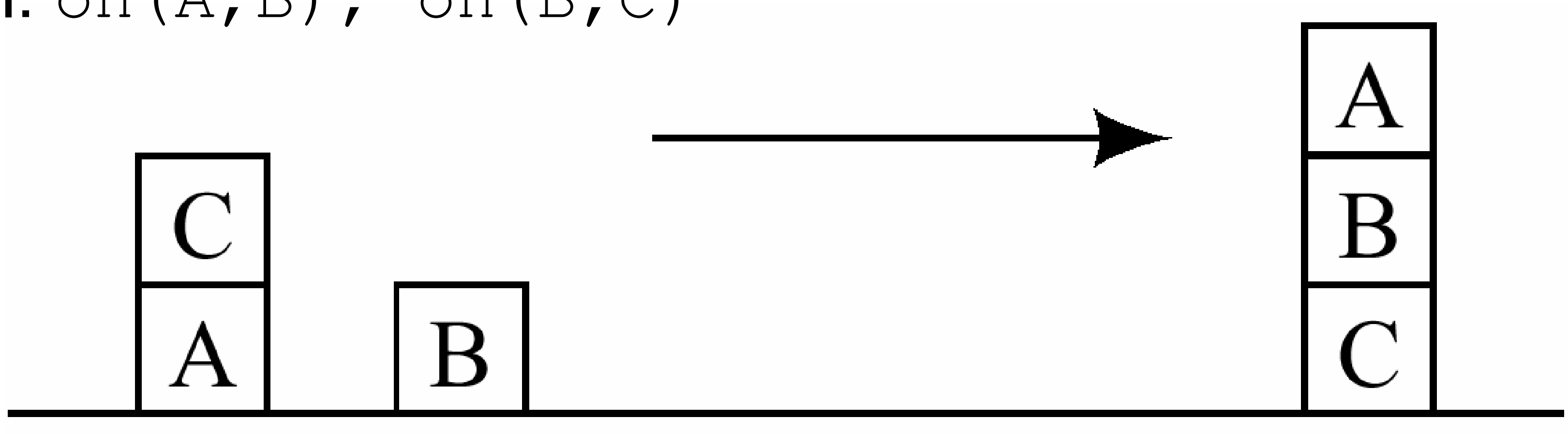
- Introduction
  - Planning vs. Problem-Solving
  - Representation in Planning Systems
- Situation Calculus
  - The Frame Problem
- STRIPS representation language
  - Blocks World
- Planning with State-Space Search
  - Progression Algorithms
  - Regression Algorithms
- Planning with Plan-Space Search
  - Partial-Order Planning
  - The Plan Graph and GraphPlan
  - SatPlan

Material from Russell & Norvig, chapters 10.3. and 11

Slides based on Slides by Russell/Norvig, Lise Getoor and Tom Lenaerts

Sussman Anomaly

- Famous example that shows that subgoals are not independent
- goal: `on(A,B), on(B,C)`



- achieve `on(B,C)` first:
  - shortest solution will just put B on top of C → subgoal has to be undone in order to complete the goal
- achieve `on(A,B)` first:
  - shortest solution will not put B on C → subgoal has do be undone later in order to complete the goal

Partial-Order Planning (POP)

- Progression and regression planning are totally ordered plan search forms
  - this means that in all searched plans the sequence of actions is completely ordered
  - Decisions must be made on how to sequence actions in all the subproblems

  → They cannot take advantage of problem decomposition
- If actions do not interfere with each other, they could be made in any order (or in parallel) → partially ordered plan
  - if a plan for each subgoal only makes minimal commitments to orders
    - only orders those actions that must be ordered for a successful completion of the plan
  - it can re-order steps later on (when subplans are combined)
  - Least commitment strategy:
    - Delay choice during search

Shoe Example

Initial State: nil
Goal State: **RightShoeOn & LeftShoeOn**

```
Action( LeftSock,
PRECOND: -
ADD:      LeftSockOn
DELETE:   -
)
```

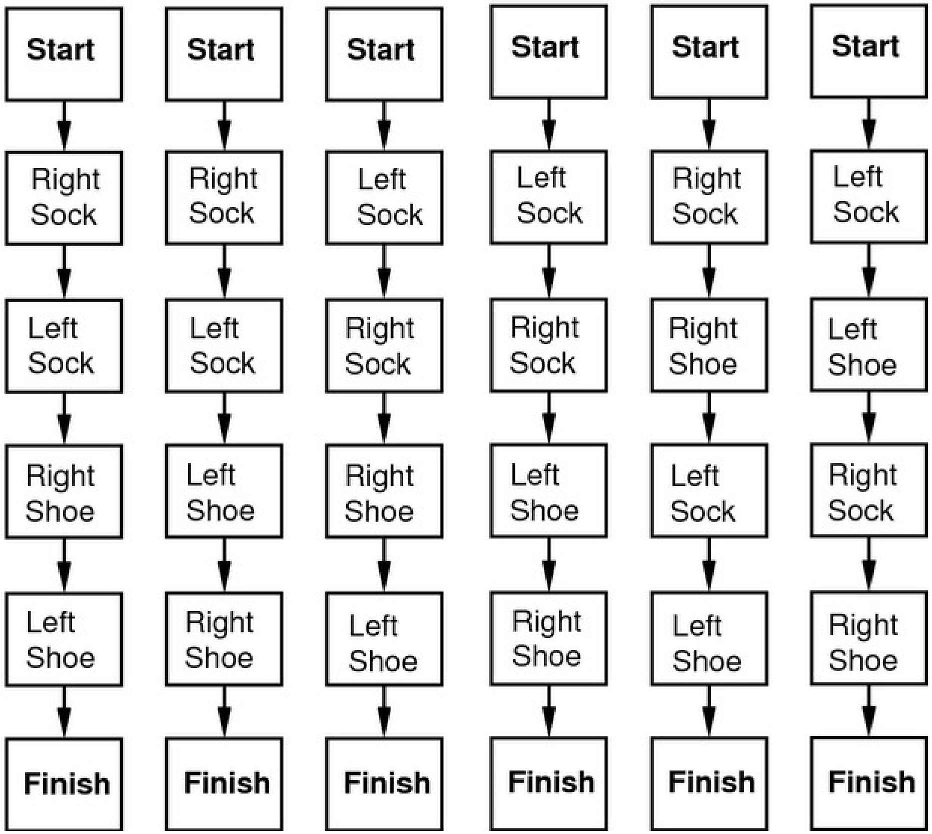
```
Action( RightSock,
PRECOND: -
ADD:      RightSockOn
DELETE:   -
)
```

```
Action( LeftShoe,
PRECOND: LeftSockOn
ADD:      LeftShoeOn
DELETE:   -
)
```

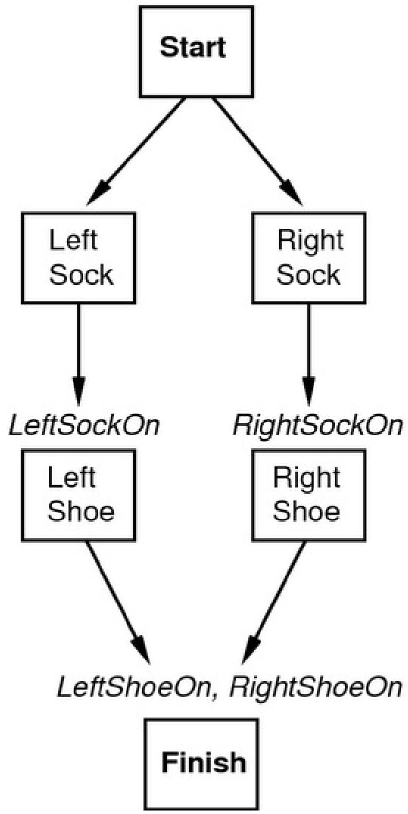
```
Action( RightShoe,
PRECOND: RightSockOn
ADD:      RightShoeOn
DELETE:   -
)
```

Shoe Example

- Total-Order Planner
  - all actions are completely ordered



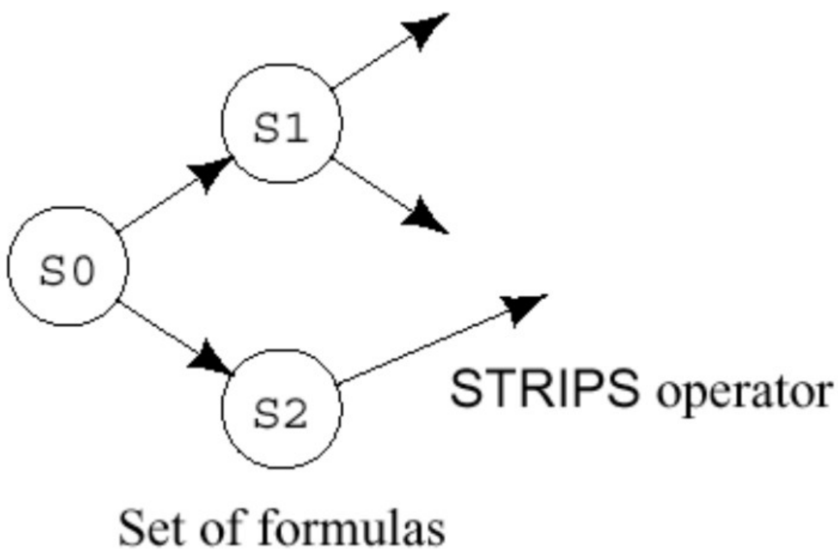
- Partial-Order Planner
  - may leave the order of some actions undetermined
  - any order is valid



State-Space vs. Plan-Space Search

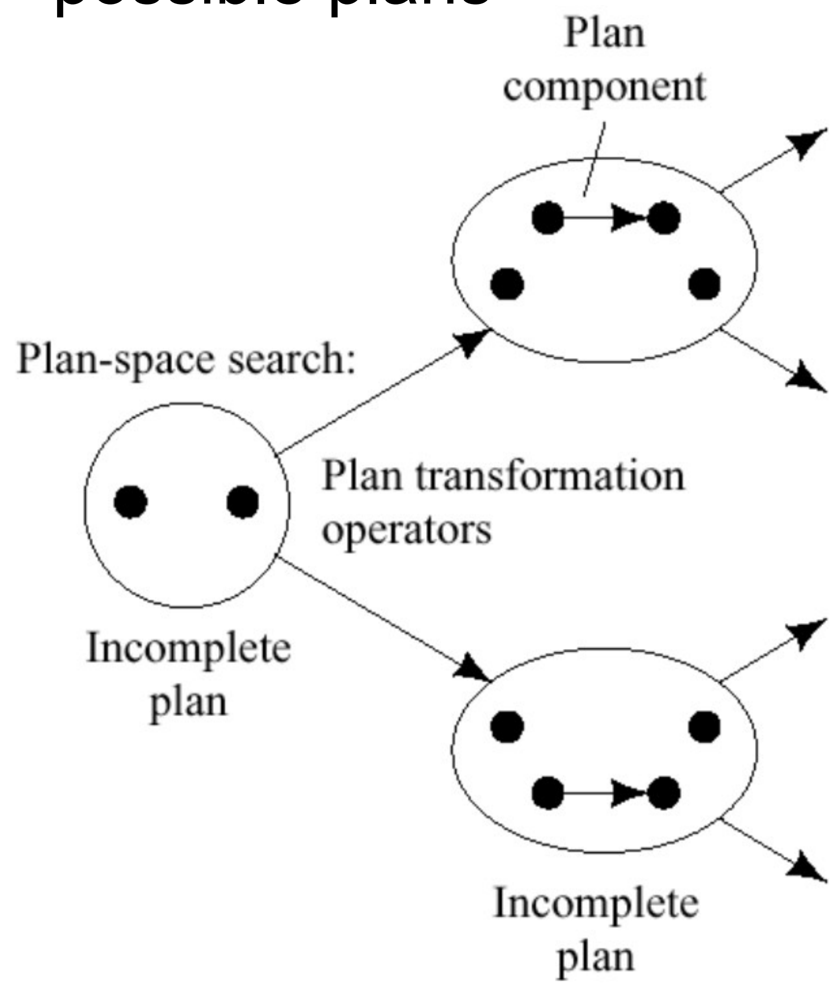
State-Space Plannning

- Search goes through possible states



Plan-Space Planning

- Search goes through possible plans



POP as a Search Problem

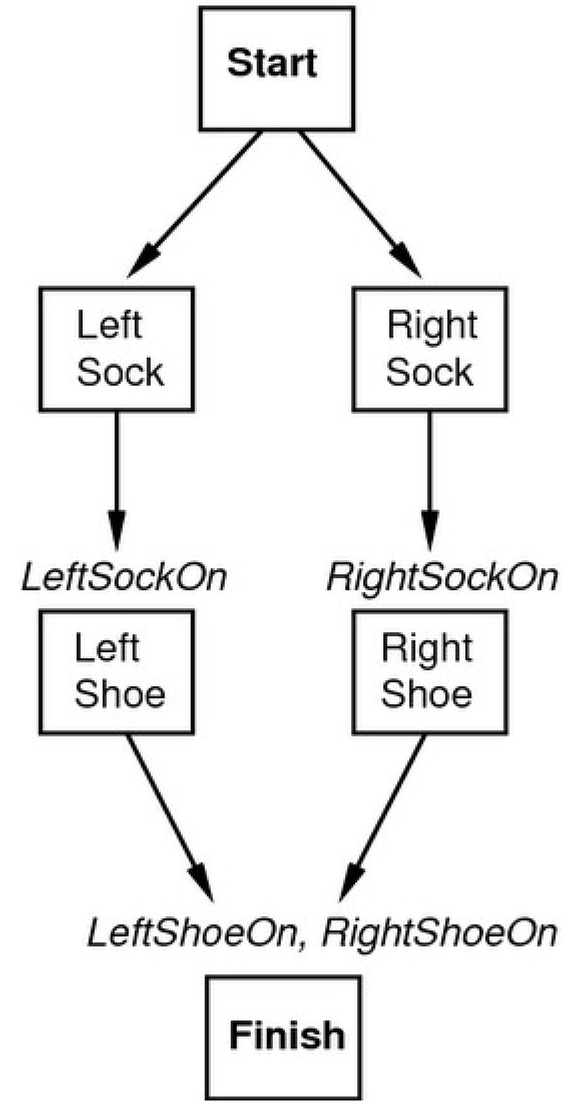
- A solution can be found by a search through Plan-Space:
  - States are (mostly unfinished) plans

Each plan has 4 components:

- A set of actions (steps of the plan)
- A set of ordering constraints: $A < B$ (A before B)
  - Cycles represent contradictions.
- A set of causal links $A \rightarrow p \rightarrow B$ (A adds p for B)
  - The plan may not be extended by adding a new action C that conflicts with the causal link.
  - An action C conflicts with causal link $A \rightarrow p \rightarrow B$
    - if the effect of C is $\neg p$ and if C could come after A and before B
- A set of open preconditions
  - Preconditions that are not achieved by action in the plan

Example of Final Plan

- Actions = {**RightSock, RightShoe, LeftSock, LeftShoe, Start, Finish**}
- Orderings = { **RightSock < RightShoe; LeftSock < LeftShoe**}
- Causal Links = { **RightSock**→RightSockOn→**RightShoe**, **LeftSock**→LeftSockOn→**LeftShoe**, **RightShoe**→RightShoeOn→**Finish**, **LeftShoe**→LeftShoeOn→**Finish**}
- Open preconditions = { }



Search through Plan-Space

- Initial State (empty plan):
  - contains only virtual **Start** and **Finish** actions
  - ordering constraint **Start** < **Finish**
  - no causal links
  - all preconditions in **Finish** are open
    - these are the original goal
- Successor Function (refining the plan):
  generates all consistent successor states
  - picks one open precondition p on an action B
  - generates one successor plan for every possible *consistent* way of choosing action that achieves p
  - a plan is consistent iff
    - there are no cycles in the ordering constraints
    - no conflicts with the causal links
- Goal test (final plan):
  - A consistent plan with no open preconditions is a solution.

Subroutines

- Refining a plan with action A, which achieves p for B:
  - add causal link $A \rightarrow p \rightarrow B$
  - add the ordering constraint $A < B$
  - add **Start** $< A$ and $A <$ **Finish** to the plan (only if A is new)
  - resolve conflicts between
    - new causal link $A \rightarrow p \rightarrow B$ and all existing actions
    - new action A and all existing causal links (only if A is new)
- Resolving a conflict between a causal link $A \rightarrow p \rightarrow B$ and an action C
  - we have a conflict if the effect of C is $\neg p$ and C could come after A and before B

  $\rightarrow$ resolved by adding the ordering constraints $C < A$ or $B < C$
    - both refinements are added (two successor plans) if both are consistent

Search through Plan-Space

- Operators on partial plans
  - Add an action to fulfill an open condition
  - Add a causal link
  - Order one step w.r.t another to remove possible conflicts
- Search gradually moves from incomplete/vague plans to complete/correct plans
- Backtrack if an open condition is unachievable or if a conflict is irresolvable
  - pick the next condition to achieve at one of the previous choice points
  - ordering of the conditions is irrelevant for completeness (the same plans will be found), but may be relevant for consistency

Executing Partially Ordered Plans

- Any particular order that is consistent with the ordering constraints is possible
  - A partial order plan is executed by repeatedly choosing any of the possible next actions.
- This flexibility is a benefit in non-cooperative environments.

Example: Spare Tire Problem

Initial State: `at(flat,axle), at(spare,trunk)`

Goal State: `at(spare,axle)`

```
Action( remove(spare,trunk),
PRECOND:  at(spare,trunk)
ADD:      at(spare,ground)
DELETE:   at(spare,trunk)
)
```

```
Action( leave-overnight,
PRECOND:  -
ADD:      -
DELETE:   at(spare,ground),
          at(spare,axle),
          at(spare,trunk),
          at(flat,ground),
          at(flat,axle)
)
```

```
Action( remove(flat,axle),
PRECOND:  at(flat,axle)
ADD:      at(flat,ground)
DELETE:   at(flat,axle)
)
```

```
Action( putOn(spare,axle),
PRECOND:  at(spare,ground),
          not(at(flat,axle)),
ADD:      at(spare,axle)
DELETE:   at(spare,ground)
)
```

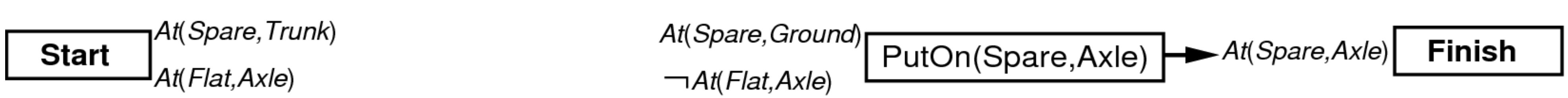
Here we need a **not**, which is not part of the original STRIPS language!

Example: Spare Tire Problem

- Initial plan:
  - Action **`start`** has the current state as effects
  - Action **`finish`** has the goal as preconditions

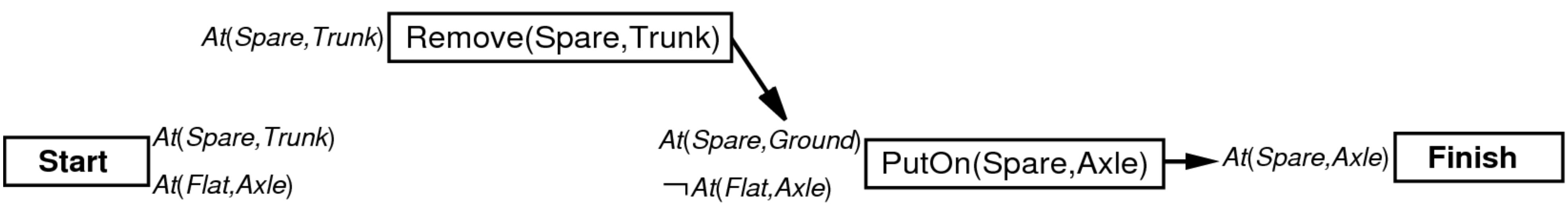
Example: Spare Tire Problem

- Action **`putOn(spare,axle)`** is the only action that achieves the goal **`at(spare,axle)`**
- the current plan is refined to one new plan:
  - **`putOn(spare,axle)`** is added to the list of actions
  - add constraints **`putOn(spare,axle) < finish`** and **`> start`**
  - add causal link **`putOn(spare,axle)→at(spare,axle)→finish`**
  - the preconditions of **`putOn(spare,axle)`** are now open



Example: Spare Tire Problem

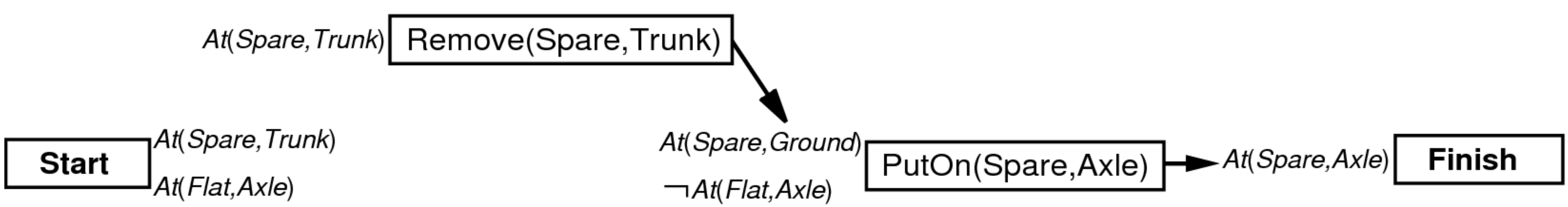
- we select the next open precondition **`at(spare,ground)`** as a goal
- only **`remove(spare,trunk)`** can achieve this goal
- the current plan is refined to a new one as before, causal links are added



Example: Spare Tire Problem

- we select the next open precondition **`not(at(flat,axle))`** as a goal
- could be achieved with two actions
  - **`leave-overnight`**
  - **`remove(flat,axle)`**

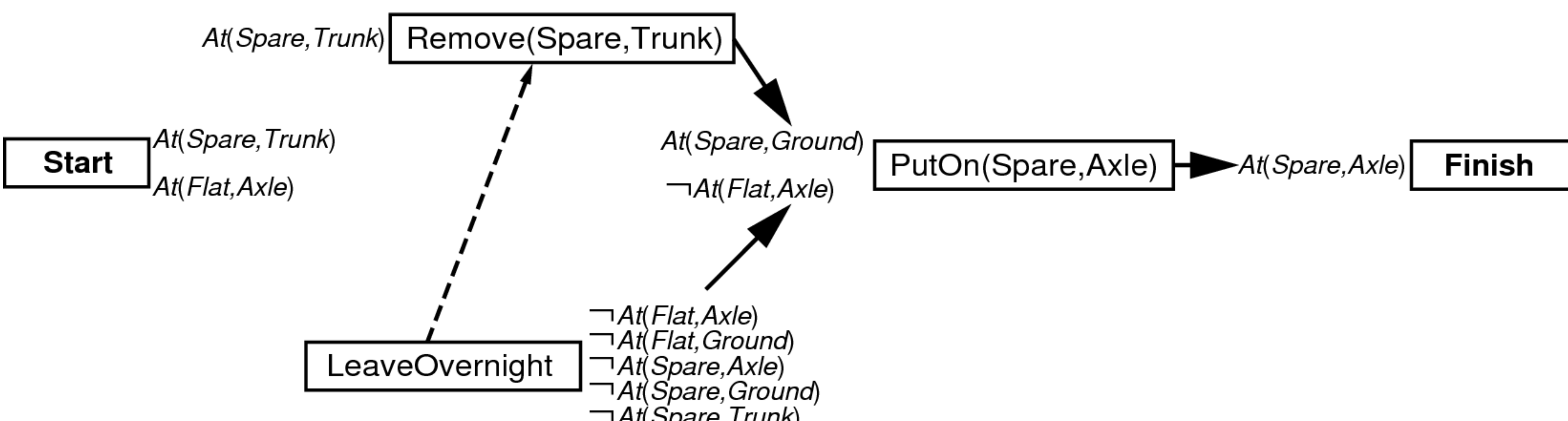
  → we have two successor plans



Example: Spare Tire Problem

Plan 1: **`leave-overnight`**

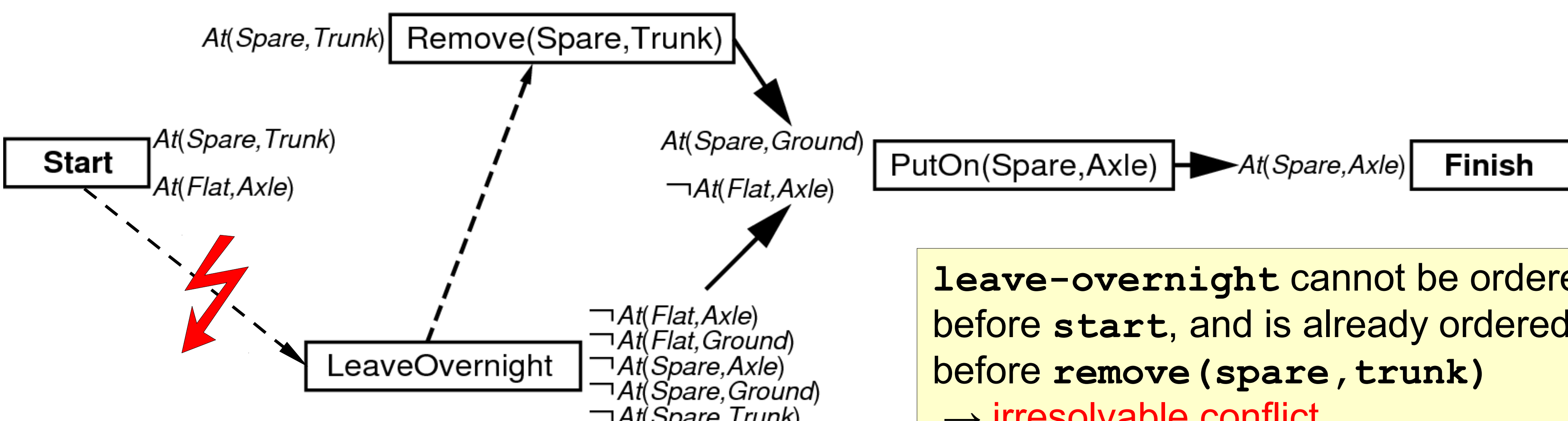
- is in conflict with the constraint
  `remove(spare,trunk)`→**`at(spare,ground)`**→**`putOn(spare,axle)`**
  → has to be ordered before **`remove(spare,trunk)`**
  - cannot be ordered after **`putOn(spare,axle)`** because it achieves its precondition
  - constraint **`leave-overnight < remove(spare,trunk)`** is added



Example: Spare Tire Problem

Plan 1: **`leave-overnight`**

- the condition **`at(spare,trunk)`** has to be achieved next
  - **`start`** is the only action that can achieve this
  - however, **`start→at(spare,trunk)→remove(spare,trunk)`** is in conflict with **`leave-overnight`**
  - this conflict cannot be resolved → backtracking

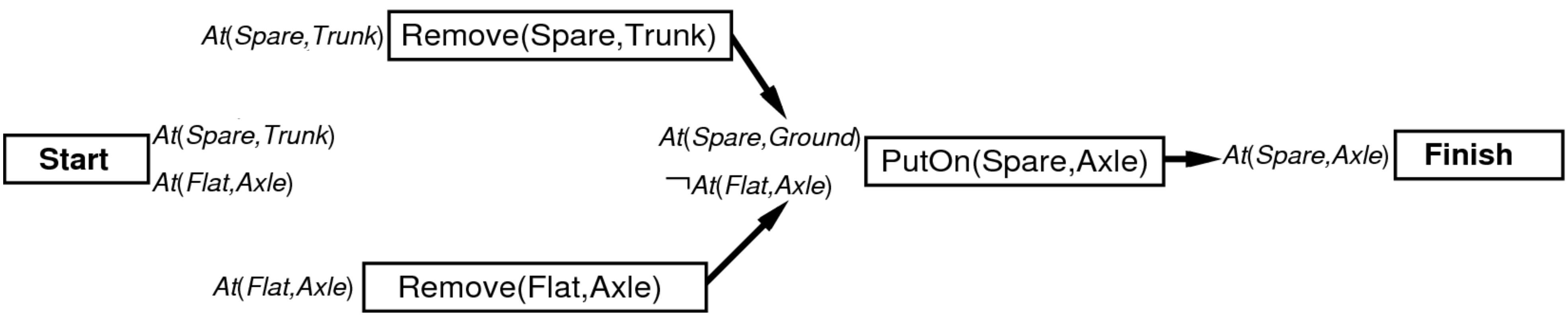


`leave-overnight` cannot be ordered before **`start`**, and is already ordered before **`remove(spare,trunk)`**
→ irresolvable conflict

Example: Spare Tire Problem

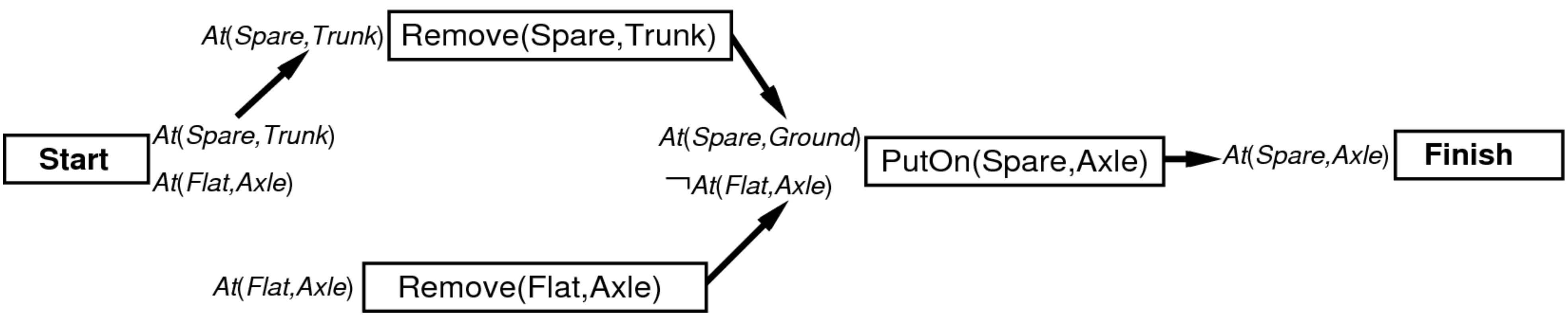
Plan 2: **`remove(flat,axle)`**

- achieves goal **`not(at(flat,axle))`**
- corresponding causal link and order relation are added
- **`at(flat,axle)`** becomes open precondition



Example: Spare Tire Problem

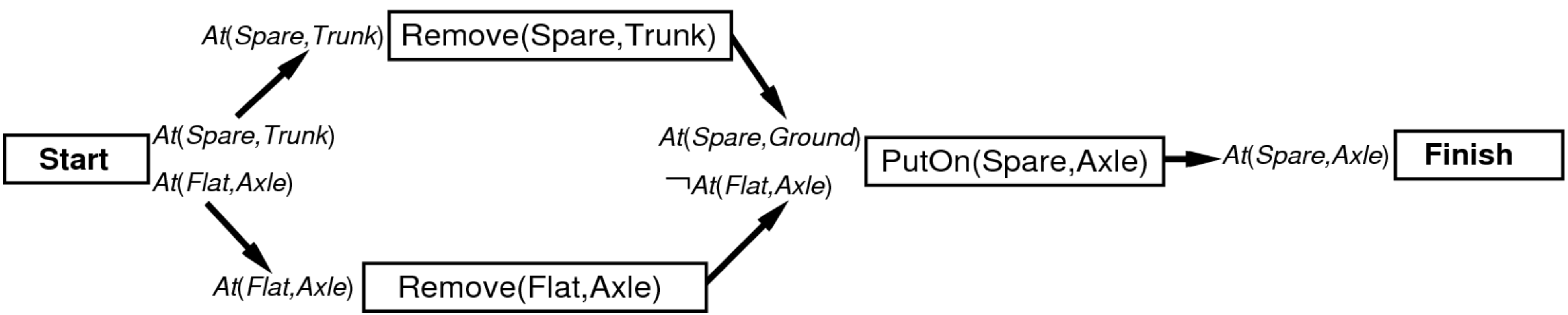
- open precondition **`at(spare,trunk)`** is selected as goal
  - action **`start`** is added
  - corresponding causal link and order relation are added



Example: Spare Tire Problem

- open precondition **`at(spare,trunk)`** is selected as goal
  - action **`start`** is added
  - corresponding causal link and order relation are added
- open precondition **`at(flat,axle)`** is selected as goal
  - action **`start`** can achieve this and is already part of the plan
  - corresponding causal link and order relation are added
- no more open preconditions remain

  → plan is completed



POP in First-Order Logic

- Operators may leave some variables unbound
- Example
  - Achieve goal **on(a,b)** with action **move(a,From,b)**
  - It remains unspecified from where block **a** should be moved (**PRECOND: on(a,From)**)

```
Action( move(Block,From,To),
PRECOND:  on(Block,From),
          clear(Block),
          clear(To),
ADD:      on(Block,To),
          clear(From),
DELETE:   on(Block,From),
          clear(To)
)
```

- Two approaches
  - Decide for one binding and backtrack later on (if necessary)
  - Defer the choice for later (least commitment)
- Problems with least commitment:
  - e.g., an action that has **on(a,From)** on its delete-list will only conflict with above if both are bound to the same variable
  - can be resolved by introducing inequality constraint.

Heuristics for Plan-Space Planning

- Not as well understood as heuristics for state-space planning
- General heuristic: number of distinct open preconditions
  - maybe minus those that match the initial state
  - underestimates costs when several actions are needed to achieve a condition
  - overestimates costs when multiple goals may be achieved with a single action
- Choosing a good precondition to refine has also a strong impact
  - select open condition that can be satisfied in the fewest number of ways
    - analogous to most-constrained variable heuristic from CSP
  - Two important special cases:
    - select a condition that cannot be achieved at all (early failure!)
    - select deterministic conditions that can only be achieved in one way

Planning Graph

- A planning graph is a special structure used to
  - achieve better heuristic estimates.
  - directly extract a solution using GRAPHPLAN algorithm
- Consists of a sequence of levels (time steps in the plan)
  - Level 0 is the initial state.
- Each level consists of a set of literals and a set of actions.
  - Literals = all those that *could* be true at that time step
    - depending on the actions executed at the preceding time step
  - Actions = all those actions that could have their preconditions satisfied at that time step
    - depending on which of the literals actually hold.
  - Only a restricted subset of possible negative interactions among actions is recorded
- Planning graphs work only for propositional problems
  - STRIPS and ADL can be propositionalized

Cake Example

- Initial state: **have(cake)**
- Goal state: **have(cake), eaten(cake)**

```
Action( eat(cake),
PRECOND:  have(cake)
ADD:      eaten(cake)
DELETE:   have(cake)
)
```

```
Action( bake(cake),
PRECOND:  not(have(cake))
ADD:      have(cake)
DELETE:   -
)
```

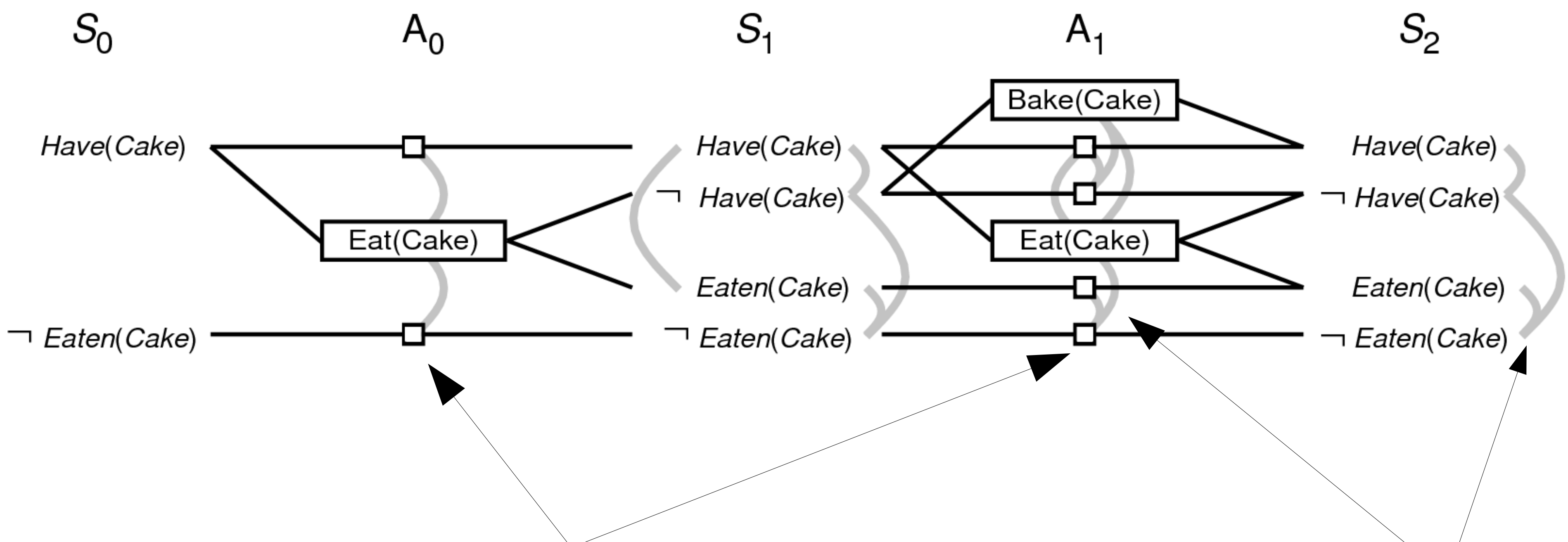
Persistence Actions

- pseudo-actions for which the effect equals the precondition
- analogous to frame axioms
- are automatically added by the planner

Mutual exclusions

- link actions or preconditions that are mutually exclusive (*mutex*)

Cake Example



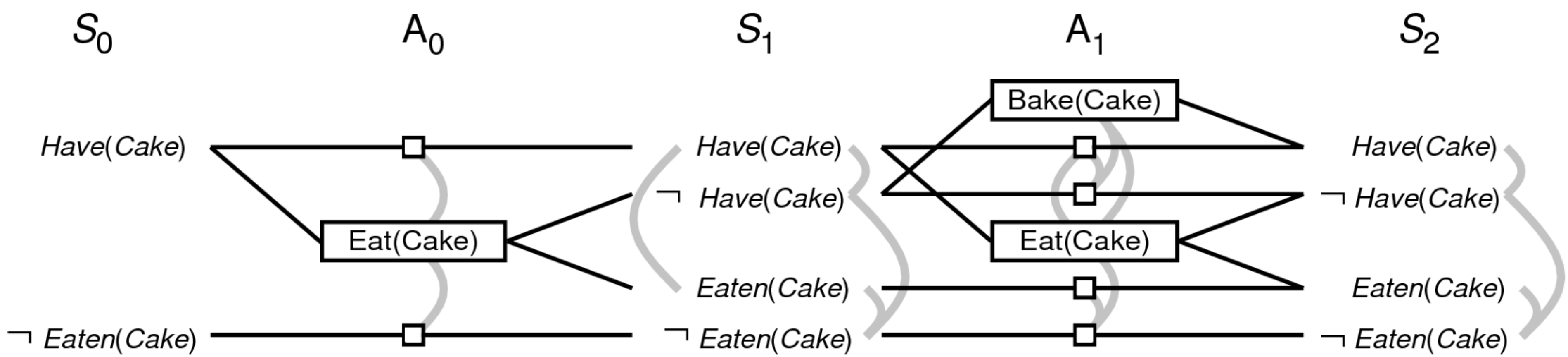
Persistence Actions (☐)

- pseudo-actions for which the effect equals the precondition
- analogous to frame axioms
- are automatically added by the planner

Mutual exclusions (—)

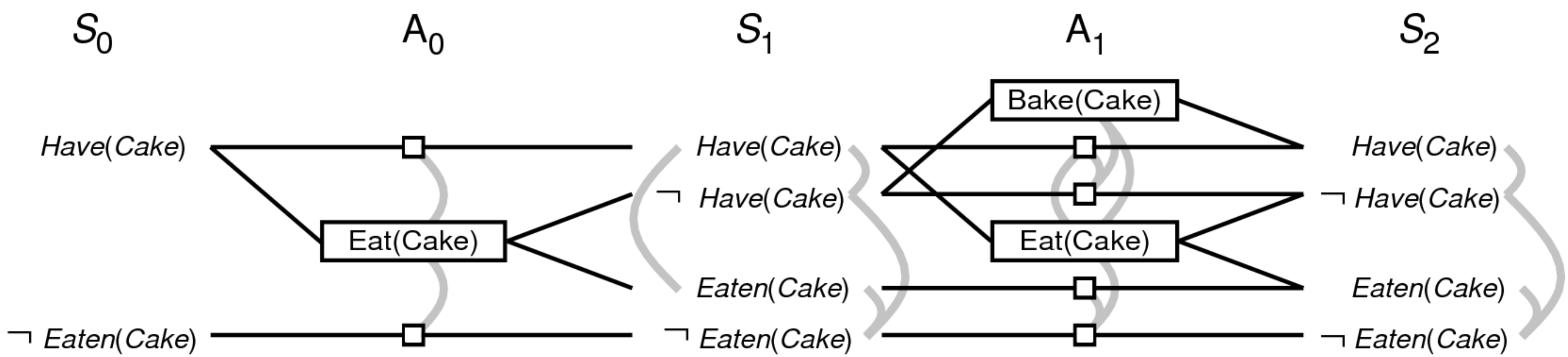
- link actions or preconditions that are mutually exclusive (*mutex*)

Cake Example



- Start at level S_0, determine action level A_0 and next level S_1
  - A_0 contains all actions whose preconditions are satisfied in the previous level S_0
  - Connect preconditions and effects of these actions
  - Inaction is represented by persistence actions
- Level A_0 contains the actions that could occur
  - Conflicts between actions are represented by mutex links

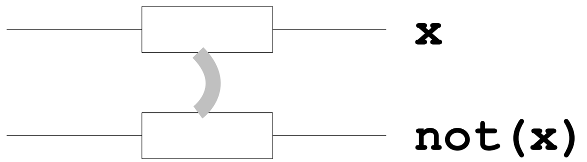
Cake Example



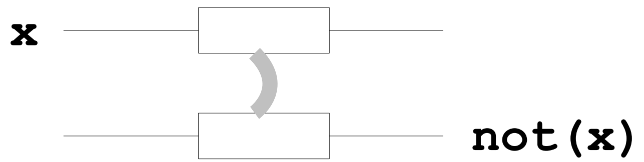
- Per construction, Level S_1 contains all literals that could result from picking any subset of actions in A_0
  - Conflicts between literals that can not occur together are represented by mutex links.
  - S_1 defines multiple possible states and the mutex links are the constraints that hold in this set of states
- Continue until two consecutive levels are identical
  - Or contain the same amount of literals (explanation later)

Mutex Relations

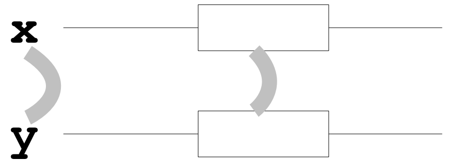
- A mutex relation holds between two actions when:
  - Inconsistent effects:
    - one action negates the effect of another.



  - Interference:
    - one of the effects of one action is the negation of a precondition of the other



  - Competing needs:
    - one of the preconditions of one action is mutually exclusive with the precondition of the other.



- A mutex relation holds between two literals when:
  - Inconsistent support:
    - If one is the negation of the other OR
    - if <u>each</u> possible action pair that could achieve the literals is mutex

Example: Spare Tire Problem

Initial State: `at(flat,axle), at(spare,trunk)`

Goal State: `at(spare,axle)`

```
Action( remove(spare,trunk),
PRECOND:  at(spare,trunk)
ADD:      at(spare,ground)
DELETE:   at(spare,trunk)
)
```

```
Action( leave-overnight,
PRECOND: -
ADD:     -
DELETE:  at(spare,ground),
         at(spare,axle),
         at(spare,trunk),
         at(flat,ground),
         at(flat,axle)
)
```

```
Action( remove(flat,axle),
PRECOND:  at(flat,axle)
ADD:      at(flat,ground)
DELETE:   at(flat,axle)
)
```

```
Action( putOn(spare,axle),
PRECOND:  at(spare,ground),
          not(at(flat,axle)),
ADD:      at(spare,axle)
DELETE:   at(spare,ground)
)
```

Here we need a **not**, which is not part of the original STRIPS language!

GRAPHPLAN Example

- S_0 consist of 5 literals (initial state and the CWA literals)

S_0

At(Spare,Trunk)

At(Flat,Axle)

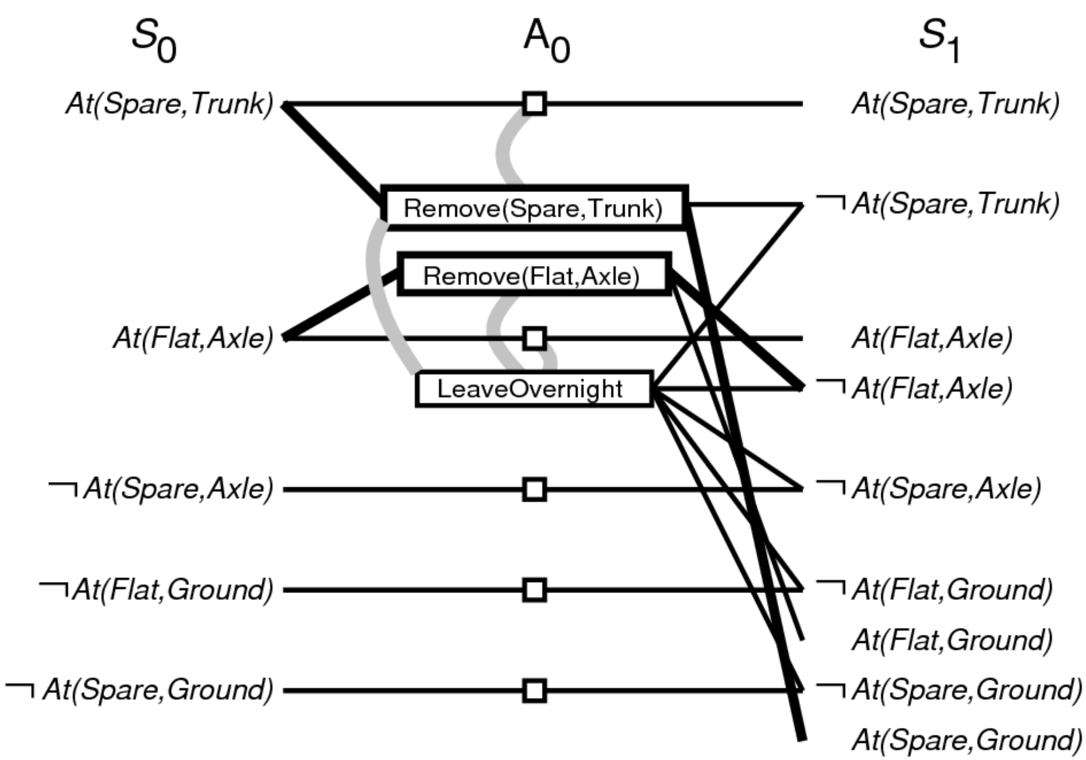
$\neg$ *At(Spare,Axle)*

$\neg$ *At(Flat,Ground)*

$\neg$ *At(Spare,Ground)*

GRAPHPLAN Example

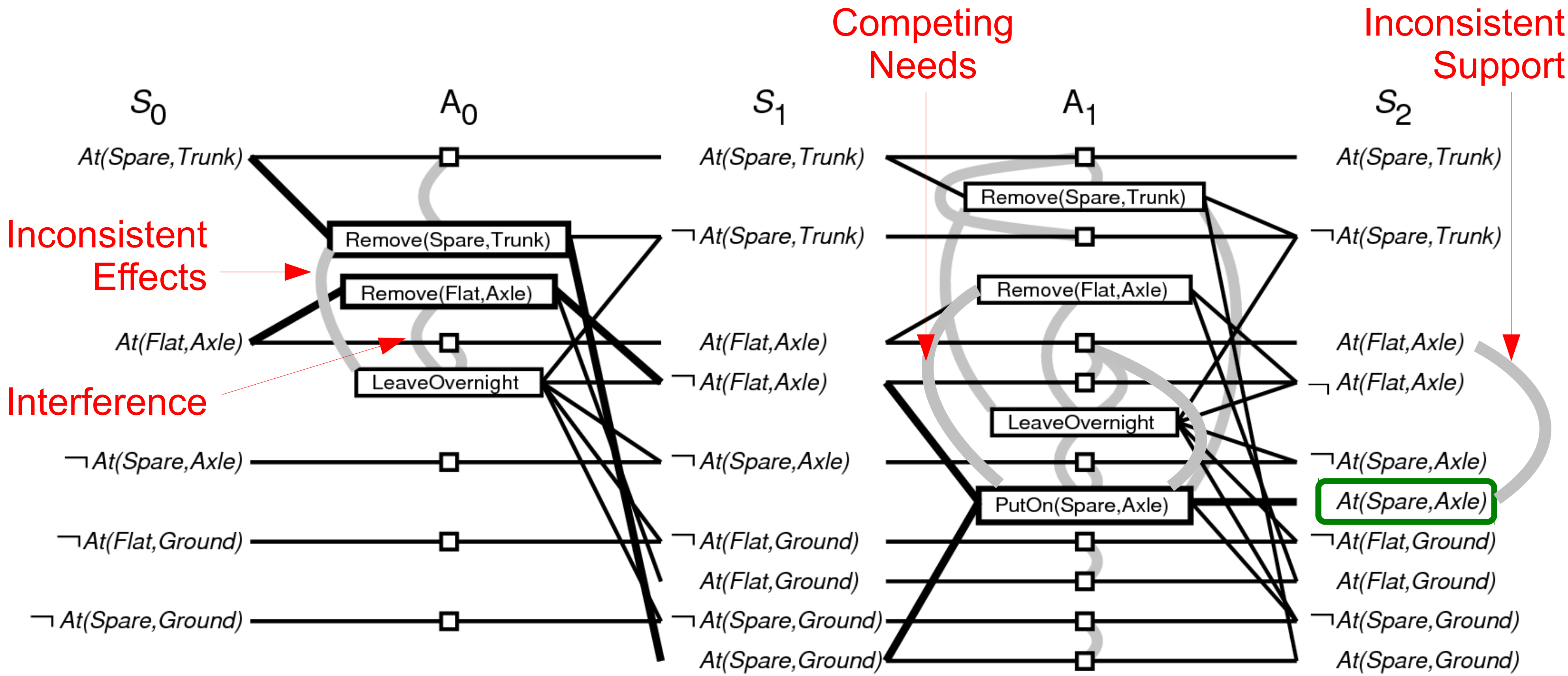
- S_0 consist of 5 literals (initial state and the CWA literals)
- EXPAND-GRAPH adds actions with satisfied preconditions
  - add the effects at level S_1
  - also add persistence actions and mutex relations



GRAPHPLAN Example

- Repeat

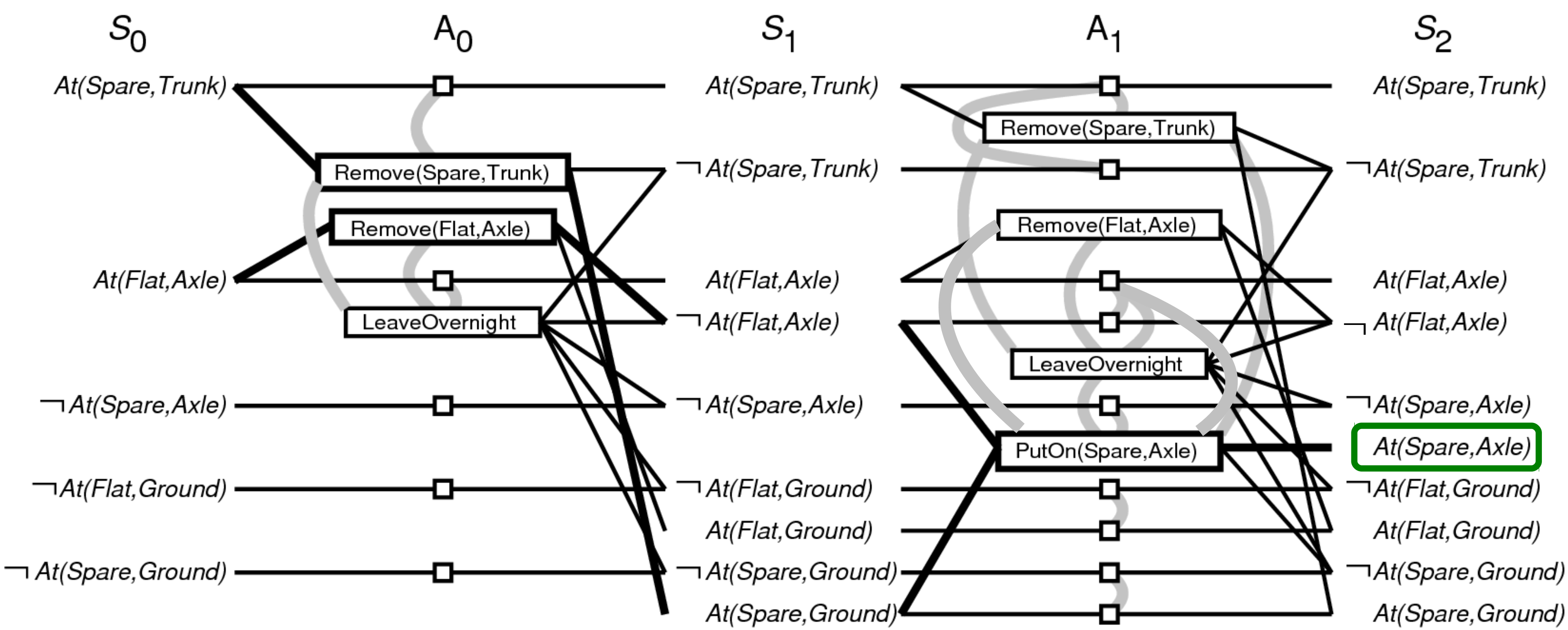
Note: Not all mutex links are shown!



GRAPHPLAN Example

- Repeat until all goal literals are pairwise non-mutex in S_i
  - If all goal literals are pairwise non-mutex, this means that a solution might exist
    - not guaranteed because only pairwise conflicts are checked

  → we need to search whether there is a solution



Deriving Heuristics from the PG

- Planning Graphs provide information about the problem
  - Example:
    - A literal that does not appear in the final level of the graph cannot be achieved by any plan
- Extraction of a serial plan
  - PG allows several actions to occur simultaneously at a level
  - can be serialized by restricting PG to one action per level
    - add mutex links between every pair of actions
  - provides a good heuristic for serial plans
- Useful for backward search
  - Any state with an unachievable precondition has cost = $+\infty$
  - Any plan that contains an unachievable precond has cost = $+\infty$
  - In general: level cost = level of first appearance of a literal
    - clearly, level cost are an admissible search heuristic
- PG may be viewed as a relaxed problem
  - checking only for consistency between pairs of actions/literals

Costs for Conjunctions of Literals

- Max-level: maximum level cost of all literals in the goal
  - admissible but not accurate
- Sum-level: sum of the level costs
  - makes the subgoal independence assumption
  - inadmissible, but works well in practice
  - Cake Example:
    - estimated costs for **`have(cake)`** $\wedge$ **`eaten(cake)`** is $0+1=1$
    - true costs are 2
  - Cake Example without action **`bake(cake)`**
    - estimated costs are the same
    - true costs are $+\infty$
- Set-level: find the level at which all literals appear and no pair has a mutex link
  - gives the correct estimate in both examples above
  - dominates max-level heuristic, works well with interactions

The GRAPHPLAN Algorithm

- Algorithm for extracting a solution directly from the PG
  - alternates solution extraction and graph expansion steps

```
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)
```

- EXTRACT-SOLUTION:
  - checks whether a plan can be found searching backwards
- EXPAND-GRAPH:
  - adds actions for the current and state literals for the next level

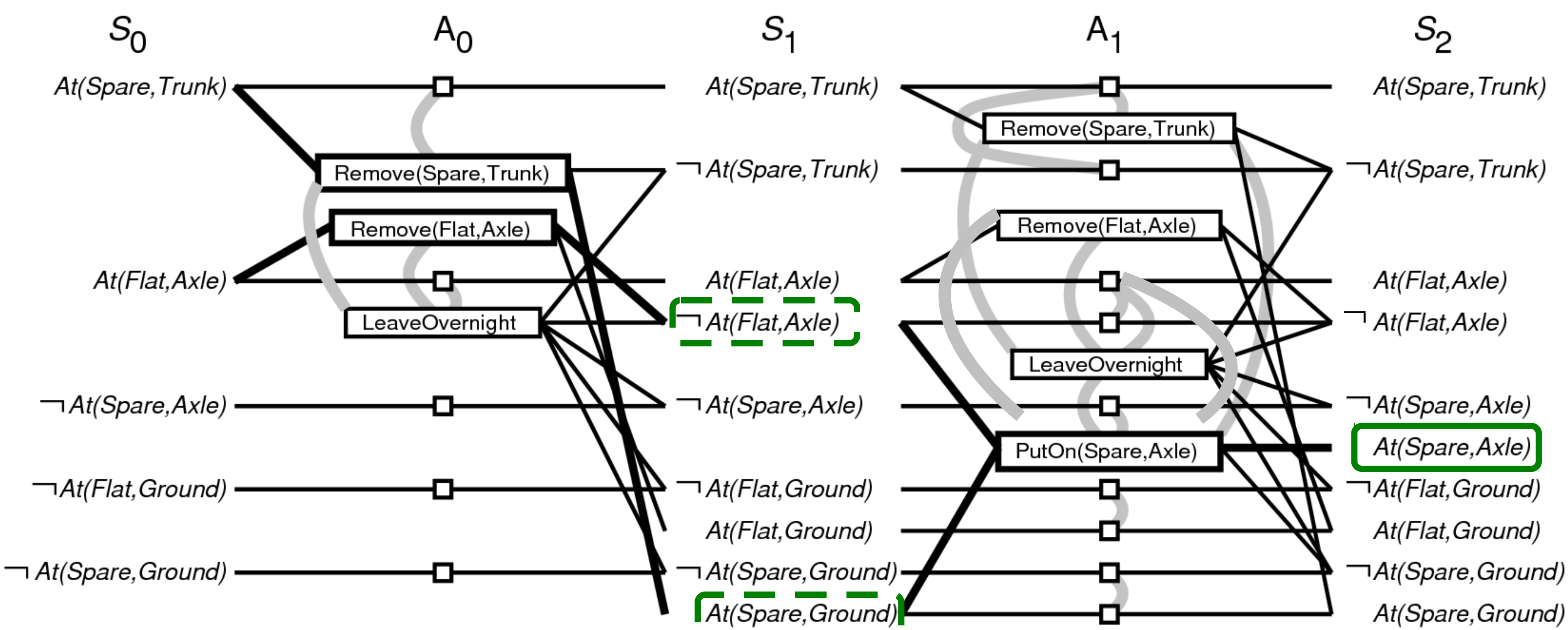
EXTRACT-SOLUTION

A state consists of
  - a pointer to a level in the planning graph
  - a set of unsatisfied goals
- Initial state
  - last level of PG
  - set of goals from the planning problem
- Actions
  - select any non-conflicting subset of the actions of A_{i-1} that cover the goals in the state
- Goal
  - success if level S_0 is reached with such with all goals satisfied
- Cost
  - 1 for each action

Could also be formulated as a Boolean CSP

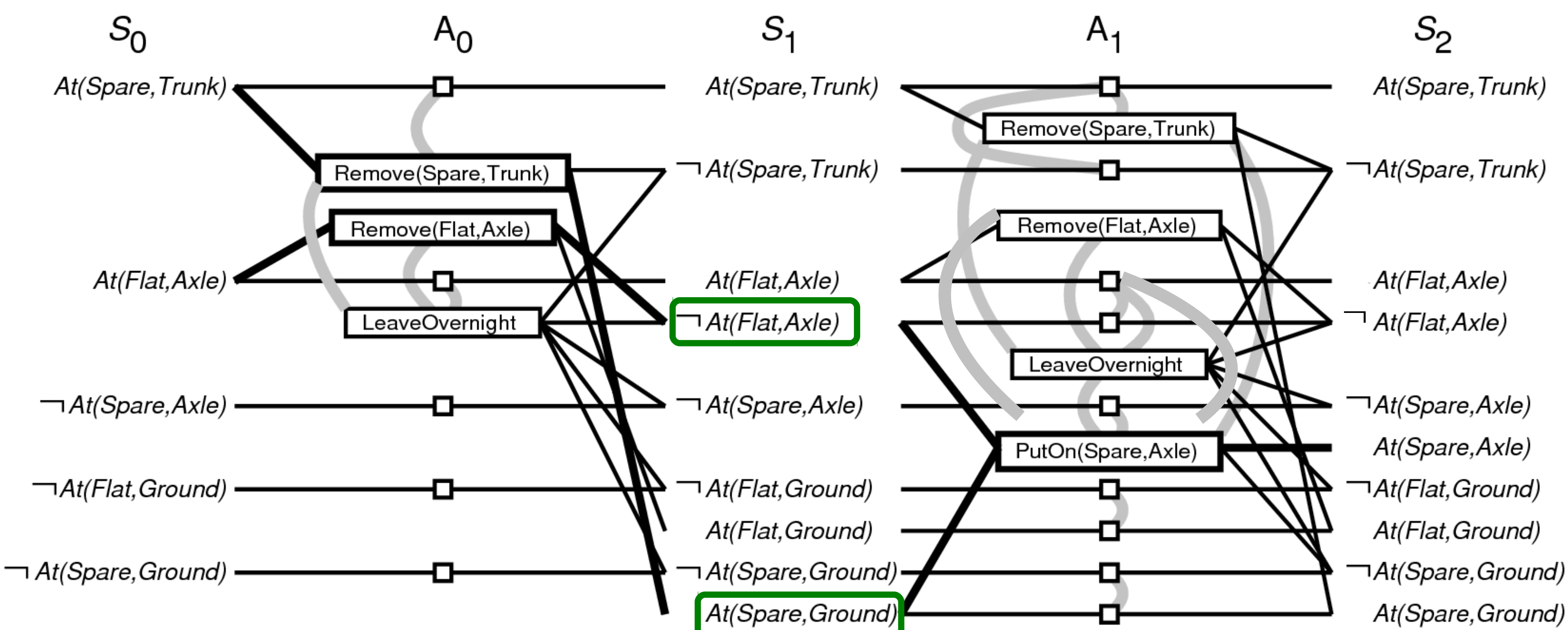
GRAPHPLAN Example

- Start with goal state **at(spare,axle)** in S_2
  → only action choice is **puton(spare,axle)** with preconditions **not(at(spare,axle))** and **at(spare,ground)** in S_1
  → two new goals in level 1



GRAPHPLAN Example

- **remove(spare,trunk)** is the only action to achieve **at(spare,ground)**
- **not(at(flat,axle))** can be achieved with **leave-overnight** and **remove(flat,axle)**
- **leave-overnight** is mutex with **remove(spare,trunk)**
  → **remove(spare,trunk)** and **remove(flat,axle)**
- preconditions are satisfied in S_0 → we're done



Termination of GRAPHPLAN

1. The planning graph converges because everything is finite
   - number of literals is monotonically increasing
     - a literal can never disappear because of the persistence actions
   - number of actions is monotonically increasing
     - once an action is applicable it will always be applicable (because its preconditions will always be there)
   - number of mutexes is monotonically decreasing
     - If two actions are mutex at one level, they are also mutex in all previous levels in which they appear together
     - inconsistent effects and interferences are properties of actions

     → if they hold once, they will always hold
     - competing needs are properties of mutexes

     → if the number of actions goes up, chances increase that there is a pair of non-mutex actions that achieve the preconditions
2. After convergence, EXTRACT-SOLUTION will find an existing solution right away or in subsequent expansions of the PG
   - more complex proof (not covered here)

SATPLAN

- Key idea:
  - translate the planning problem into propositional logic
  - similar to situation calculus, but all facts and rules are ground
    - the same literal in different situations is represented with two different propositions (we call them propositions at a depth i)
  - actions are also represented as propositions
  - rules are used to derive propositions of depth i+1 from actions and propositions of depth i
- Goal:
  - find a true formula consisting of propositions of the initial state, propositions of the goal state, and $\underbrace{\text{some action propositions}}_{\text{the plan!}}$
- Method:
  - use a satisfiability solver with iterative deepening on the depth
    - first try to prove the goal in depth 0 (initial state)
    - then try to prove the goal in depth 1
    - .... until a solution is found in depth *n*

Key Problem

- Complexity
  - In the worst case, a proposition has to be generated
    - for each of a actions with
    - each of o possible objects in the n arguments
    - for a solution depth d

  → maximum number of propositions is $d \cdot a \cdot o^n$
  - the number of rules is even larger

Solution Attempt: Symbol Splitting
  - a possible solution is to convert each n-ary relation into n binary relations
    - "the i-th argument of relation r is y"
  - this will also reduce the size of the knowledge base because arguments that are not used can be omitted from the rules
  - Drawback: multiple instances of the same rule get mixed up

    → no two actions of same type at the same time step
- Nevertheless, SatPlan is very competitive