Artificial Intelligence
Unit 4-1 Planning

2020-2021 Odd BE CSE VII semester

Engels. R

Unit 4 – Planning

- **Planning and Learning:**
  - **Planning with State Space Search:**
    - Partial Order Planning
    - Planning Graphs
    - Examples
- Blended/Self learning
  - **Forms of Learning**
    - ***Inductive Learning***
    - ***Explanation Based Learning***
    - ***Statistical Learning***
    - ***Learning With Complete Data***

Background

- Focus
  - The focus here is **deterministic planning**
    - Environment is fully observable
    - Results of actions is deterministic
  - Relaxing the above requires dealing with uncertainty
    - Problem types: sensor-less, contingency, exploration
- Planning ‘communities’ in AI
  - **Logic-based:** Reasoning About Actions & Change
  - Less formal representations: **Classical AI Planning**
  - **Uncertainty (UAI):** Graphical Models such as
    - Markov Decision Processes (MDP), Partially Observable MDPs, etc.

Actions, events, and change

- Planning requires a representation of time
  - to express & reason about sequences of actions
  - to express the effects of actions on the world
- Propositional Logic
  - does not offer a representation for time
  - Each action description needs to be repeated for each step
- Situation Calculus
  - Is based on FOL
  - Each time step is a ‘situation’
  - Allows to represent plans and reason about actions & change

‘Famous’ Problems

- Frame problem
  - **Representing all things that stay the same from one situation to the next**
  - Inferential and representational
  - What happens to a gold bar placed on a slide?
- Qualification problem
  - Defining the circumstances under which **an action is guaranteed to work**
  - Example: what if the gold is slippery or nailed down, etc.
- Ramification problem
  - Proliferation of **implicit consequences of actions** as actions may have secondary consequences
  - Examples: How about the dust on the gold?

Planning Languages

- Languages must represent..
  - **S**tates
  - **G**oals
  - **A**ctions
- Languages must be
  - Expressive for ease of representation
  - Flexible for manipulation by algorithms

State Representation

- A state is represented with a conjunction of **positive literals**
  - fluents that are ground, functionless atoms
  - a **fluent** is a condition that can change over time
- Using
  - Logical Propositions: ***Poor*** $\wedge$ ***Unknown***
  - FOL literals: ***At(Plane1,SFO)*** $\wedge$ ***At(Plane2,JFK)***
- FOL literals must be ground & function-free
  - **Not allowed**: **At(x,y)** [non-ground]or **At(Father(Fred),Sydney)**[Function], ¬poor (negation, false fluent)
- Closed World Assumption
  - What are not stated are assumed false

Action Representation

- A set of ground (variable-free) actions can be represented by a single **action schema**
  - The schema is a **lifted** representation
  - it lifts the level of reasoning **from propositional logic to** a **restricted subset of first-order logic**

Action Representation

- Action Schema
  - Action name
  - Preconditions
  - Effects
- Example

  Action(Fly(p,from,to),

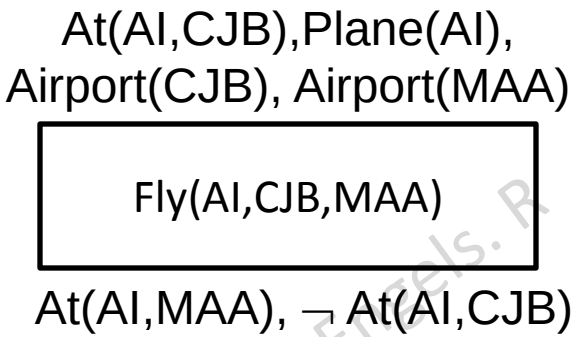
  PRECOND: At(p,from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)

  EFFECT: ¬At(p,from) ∧ At(p,to))
- Sometimes, Effects are split into ADD list and DELETE list

  RESULT(s, a) = [s − DEL(a)] ∪ ADD(a)

  s- state, a-action, s'-new state

  DEL(a) – actions available at s, ADD(a) – actions available at s'



Applying an Action

- **Find a substitution list** θ for the variables
  - of all the precondition literals
  - with (a subset of) the literals in the current state description
- **Apply** the substitution to the propositions in the effect list
- **Add the result** to the current state description to generate the new state
- Example:
  - Current state: At(P1,JFK) $\wedge$ At(P2,SFO) $\wedge$ Plane(P1) $\wedge$ Plane(P2) $\wedge$ Airport(JFK) $\wedge$ Airport(SFO)
  - It satisfies the precondition with θ={**p/P1**,**from/JFK**, **to/SFO**)
  - Thus the action Fly(P1,JFK,SFO) is applicable
  - The **new current state** is: At(P1,SFO) $\wedge$ At(P2,SFO) $\wedge$ Plane(P1) $\wedge$ Plane(P2) $\wedge$ Airport(JFK) $\wedge$ Airport(SFO)

Goal & Action Representation

- Goal is a partially specified state
- A proposition satisfies a goal if it contains all the atoms of the goal and possibly others..
  - Example: Rich $\wedge$ Famous $\wedge$ Miserable **satisfies** the goal Rich $\wedge$ Famous

Languages for Planning Problems

- STRIPS
  - Stanford Research Institute Problem Solver
  - Historically important
- ADL
  - Action Description Languages
- **PDDL**
  - ***Planning Domain Definition Language***
  - Describes the four things we need to define a search problem:
  - the **initial** state, the **actions** that are available in a state, the **result** of applying an action, and the **goal** test

Example: Air Cargo

- Initial state, Goal State
- Actions: Load, Unload, Fly

$$
\begin{aligned}
&Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK) \\
&\quad \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2) \\
&\quad \land Airport(JFK) \land Airport(SFO)) \\
&Goal(At(C_1, JFK) \land At(C_2, SFO)) \\
&Action(Load(c, p, a), \\
&\quad \text{PRECOND: } At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a) \\
&\quad \text{EFFECT: } \neg At(c, a) \land In(c, p)) \\
&Action(Unload(c, p, a), \\
&\quad \text{PRECOND: } In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a) \\
&\quad \text{EFFECT: } At(c, a) \land \neg In(c, p)) \\
&Action(Fly(p, from, to), \\
&\quad \text{PRECOND: } At(p, from) \land Plane(p) \land Airport(from) \land Airport(to) \\
&\quad \text{EFFECT: } \neg At(p, from) \land At(p, to))
\end{aligned}
$$

A PDDL description of an air cargo transportation planning problem:
Loading and unloading cargo and
flying it from place to place

Solution?

Start with [Load(C1, P1, SFO)….]

Example: Air Cargo

$Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK)$
$\land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2)$
$\land Airport(JFK) \land Airport(SFO))$
$Goal(At(C_1, JFK) \land At(C_2, SFO))$
$Action(Load(c, p, a),$
PRECOND: $At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
EFFECT: $\neg At(c, a) \land In(c, p))$
$Action(Unload(c, p, a),$
PRECOND: $In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
EFFECT: $At(c, a) \land \neg In(c, p))$
$Action(Fly(p, from, to),$
PRECOND: $At(p, from) \land Plane(p) \land Airport(from) \land Airport(to)$
EFFECT: $\neg At(p, from) \land At(p, to))$

A PDDL description of an air cargo transportation planning problem:
Loading and unloading cargo and
flying it from place to place

The following plan is a solution to the problem:

$$[Load(C_1, P_1, SFO), Fly(P_1, SFO, JFK), Unload(C_1, P_1, JFK),$$
$$Load(C_2, P_2, JFK), Fly(P_2, JFK, SFO), Unload(C_2, P_2, SFO)] .$$

Example: Spare Tire Problem

$$Init(Tire(Flat) \land Tire(Spare) \land At(Flat, Axle) \land At(Spare, Trunk))$$
$$Goal(At(Spare, Axle))$$
$$Action(Remove(obj, loc),$$
$$\quad \text{PRECOND: } At(obj, loc)$$
$$\quad \text{EFFECT: } \neg At(obj, loc) \land At(obj, Ground))$$
$$Action(PutOn(t, Axle),$$
$$\quad \text{PRECOND: } Tire(t) \land At(t, Ground) \land \neg At(Flat, Axle)$$
$$\quad \text{EFFECT: } \neg At(t, Ground) \land At(t, Axle))$$
$$Action(LeaveOvernight,$$
$$\quad \text{PRECOND:}$$
$$\quad \text{EFFECT: } \neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk)$$
$$\qquad \land \neg At(Flat, Ground) \land \neg At(Flat, Axle) \land \neg At(Flat, Trunk))$$

Solution?

A PDDL description of changing a flat tire problem:

Goal is to have a good spare tire properly mounted onto the car's axle,
where the initial state has (a) a flat tire on the axle and (b) a good spare tire in the trunk

Four actions:

1. Removing spare from trunk,
2. Removing flat tire from axle,
3. Putting spare on axle, and
4. Leaving car unattended overnight (which will result in both spare and flat tires stolen)

Example: Spare Tire Problem

$$Init(Tire(Flat) \land Tire(Spare) \land At(Flat, Axle) \land At(Spare, Trunk))$$
$$Goal(At(Spare, Axle))$$
$$Action(Remove(obj, loc),$$
$$\quad \text{PRECOND: } At(obj, loc)$$
$$\quad \text{EFFECT: } \neg At(obj, loc) \land At(obj, Ground))$$
$$Action(PutOn(t, Axle),$$
$$\quad \text{PRECOND: } Tire(t) \land At(t, Ground) \land \neg At(Flat, Axle)$$
$$\quad \text{EFFECT: } \neg At(t, Ground) \land At(t, Axle))$$
$$Action(LeaveOvernight,$$
$$\quad \text{PRECOND:}$$
$$\quad \text{EFFECT: } \neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk)$$
$$\qquad \land \neg At(Flat, Ground) \land \neg At(Flat, Axle) \land \neg At(Flat, Trunk))$$

Solution

$$[Remove(Flat, Axle), Remove(Spare, Trunk), PutOn(Spare, Axle)]$$

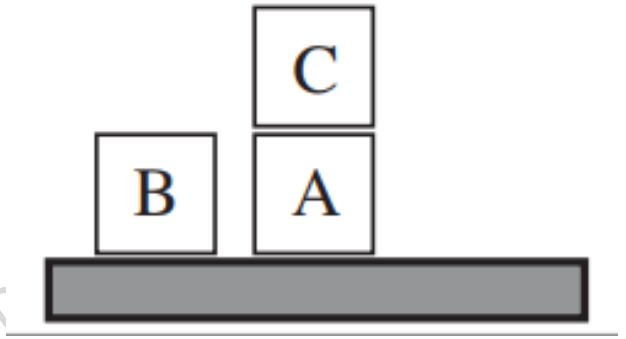
A PDDL description of changing a flat tire problem:
Goal is to have a good spare tire properly mounted onto the car's axle,
where the initial state has (a) a flat tire on the axle and (b) a good spare tire in the trunk

Example: Blocks World

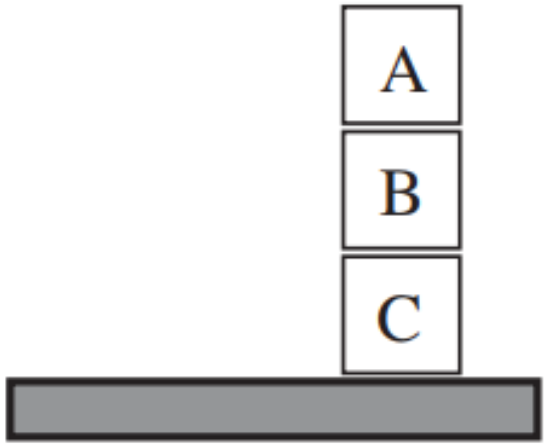
- **Initial state**, **Goal**
- Actions:
- ***Move*(b,x,y)** (b-block, x-from, y – to)
- ***MoveToTable***(b,x)
- Describe init, goal and actions(with pre-condition and effects
- Then give the solution plan

A PDDL description of the blocks-world problem:
Goal is to build one or more stacks of blocks,

where the blocks can be stacked, but only one block can fit directly on top of another, or kept on the table



initial state



goal state

Example: Blocks World

- Initial state, Goal
- Actions: *Move*(b,x,y), *MoveToTable*(b,x)

$$
\begin{aligned}
&Init(On(A, Table) \wedge On(B, Table) \wedge On(C, A) \\
&\quad \wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(B) \wedge Clear(C)) \\
&Goal(On(A, B) \wedge On(B, C)) \\
&Action(Move(b, x, y), \\
&\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge \\
&\qquad (b \neq x) \wedge (b \neq y) \wedge (x \neq y), \\
&\quad \text{EFFECT: } On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)) \\
&Action(MoveToTable(b, x), \\
&\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Block(b) \wedge (b \neq x), \\
&\quad \text{EFFECT: } On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))
\end{aligned}
$$

A PDDL description of the blocks-world problem:
Goal is to build one or more stacks of blocks,

where the blocks can be stacked, but only one block can fit directly on top of another, or kept on the table

Solution?

Example: Blocks World

- Initial state, Goal
- Actions: *Move*(b,x,y), *MoveToTable*(b,x)

$$
\begin{aligned}
&Init(On(A, Table) \wedge On(B, Table) \wedge On(C, A) \\
&\quad \wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(B) \wedge Clear(C)) \\
&Goal(On(A, B) \wedge On(B, C)) \\
&Action(Move(b, x, y), \\
&\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge \\
&\qquad (b \neq x) \wedge (b \neq y) \wedge (x \neq y), \\
&\quad \text{EFFECT: } On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)) \\
&Action(MoveToTable(b, x), \\
&\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Block(b) \wedge (b \neq x), \\
&\quad \text{EFFECT: } On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))
\end{aligned}
$$

A PDDL description of the blocks-world problem:
Goal is to build one or more stacks of blocks,

where the blocks can be stacked, but only one block can fit directly on top of another, or kept on the table

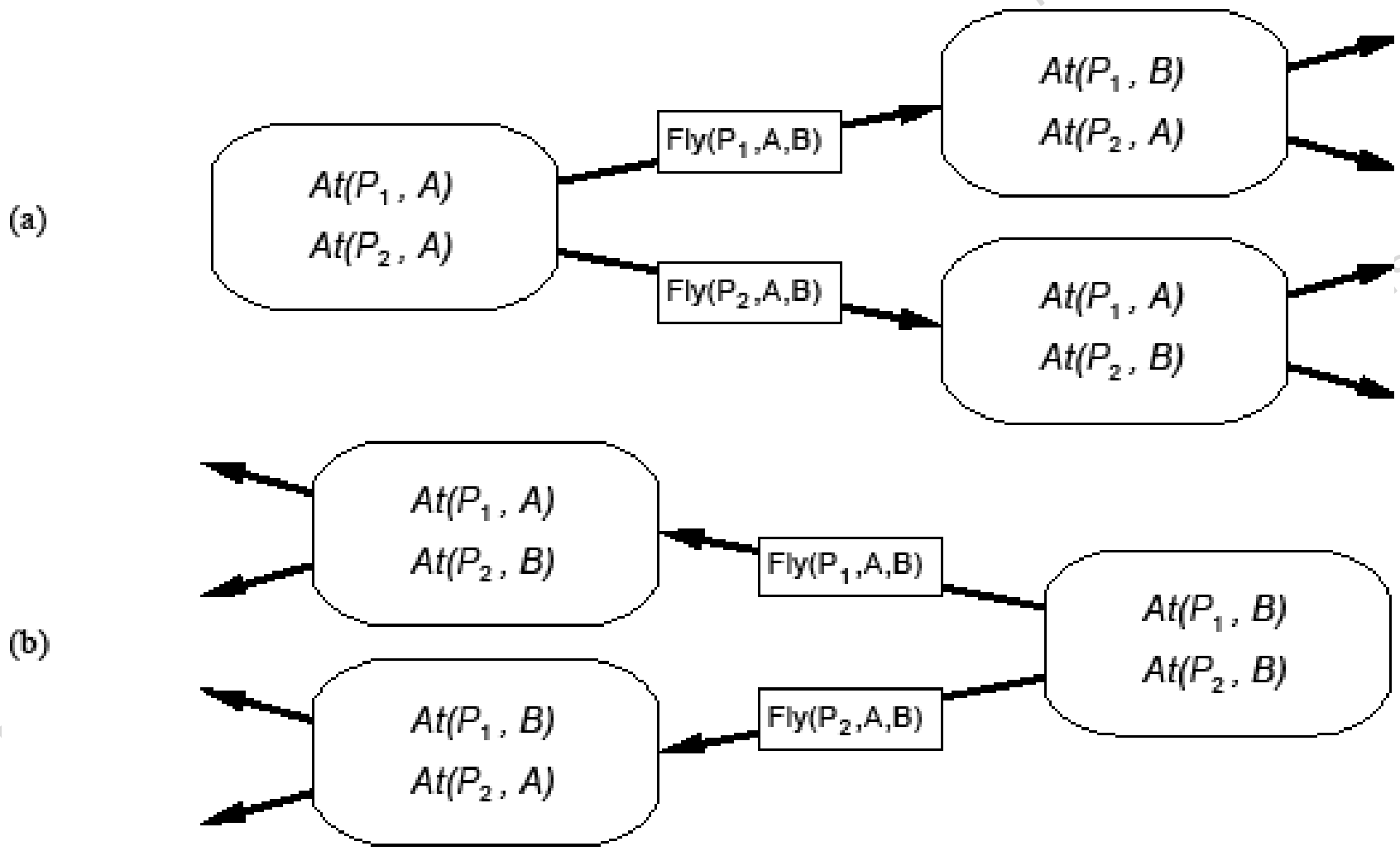
Notice how the solution plan finding becomes **extremely straight forward** with the **appropriate representation**

One solution is the sequence:
[**MoveToTable** (C,A), **Move**(B, Table, C), **Move**(A, Table,B)]

State-Space Search

- We want to see if the planning process can be solved using Search Algorithms
  - Problem must be formulated suitable for search
  - Define initial, goal, transition model, state space tree and search strategy
- 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

Forward and Backward State Space Search



Two approaches to searching for a plan.
(a) Forward (progression) search, through the space of states, ***starting in the initial state*** and using the problem's actions to ***search forward for a member of the set of goal states***

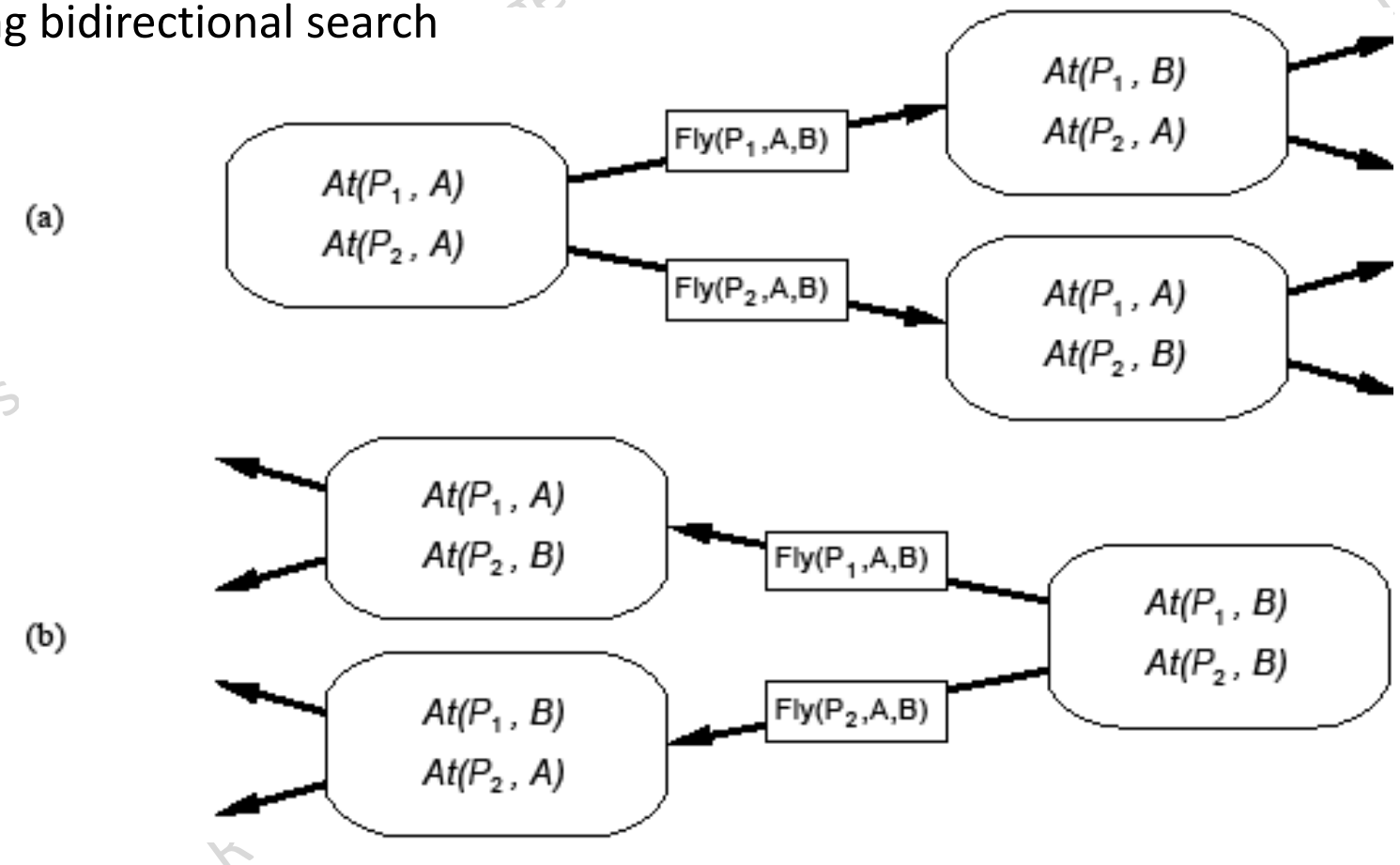
(b) Backward (regression) search through sets of relevant states, ***starting at the set of states representing the goal*** and using the inverse of the actions to ***search backward for the initial state***

State-Space Search (3)

- Remember that the language has no **functions** symbols
- Thus number of states is finite
- And we can use any **complete** search algorithm (e.g., A*)
  - [Completeness – does the algorithm always find a solution if one exists]
  - We need an admissible heuristic
  - The solution is a path, a sequence of actions: **total-order planning**
- Problem: Space and time complexity
  - only positive preconditions and
  - only one literal effect
- With the above identified we can use any appropriate search algorithm for planning.

STRIPS & State-Space Search

- STRIPS : Easy to focus on ‘relevant’ propositions and
  - Work backward from goal (using **Effects**)
  - Work forward from initial state (using **PRECONDITIONS**)
  - Facilitating bidirectional search



Relevant and Consistent Actions

- 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**
- The purpose of applying an action is to '**achieves a desired literal**'
  - 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**

Backward State-Space Search

- Given
  - A goal *G* description
  - An action *A* that is relevant and consistent
- Generate a predecessor state where
  - Positive effects (literals) of *A* in *G* are deleted
  - Precondition literals of *A* are added unless they already appear
  - Substituting any variables in A's effects to match literals in G
  - Substituting any variables in A's preconditions to match substitutions in A's effects
- Repeat until predecessor description matches initial state

Heuristic to Speed up Search

- We can use A*, but we need an admissible heuristic
  - Must not overestimate cost from current s to goal g
  - Such as the straight line distance (h_{SLD}) we used earlier)
- Such an admissible heuristic can be derived with
  - by defining a **relaxed problem** whose solution can be used as admissible heuristic for the given problem
- Think of a search problem as a graph
  - where the nodes are states and the edges are actions
- The problem is to find a path connecting the initial state to a goal state
- There are two ways we can relax this problem to make it easier:
  - by **adding more edges** to the graph, making it strictly easier to find a path, or
  - by **grouping multiple nodes together**, forming an abstraction of the state space that has fewer states, and thus is easier to search

Heuristic to Speed up Search

Approach

1. Divide-and-conquer: sub-goal independence assumption
   - Problem relaxation is done through one of the following
   - By removing
2. … all preconditions
3. … all preconditions and negative effects
4. … negative effects only: Empty-Delete-List

1. Subgoal Independence Assumption

- A key idea in defining heuristics is **decomposition**:
  The cost of solving a conjunction of subgoals is the sum of the costs of solving each subgoal independently
- Optimistic
  - Assume that there are no subplans that interact negatively when put-together
  - 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 action becomes applicable in every state, and
  - Every literal in the goal is achievable in one step
    - Any single goal fluent can be achieved in one step
    - if there is an applicable action—if not, the problem is impossible
- 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 (1)
  - Some action may remove the effect of another action (2)
  - For many problems an accurate heuristic is obtained by considering (1) and ignoring (2)

3. Remove Preconditions & Negative Effects

- Considers only positive interactions among actions to achieve multiple subgoals
- The minimum number of actions required is
  - Sum of the union of the actions' positive effects that satisfy the goal

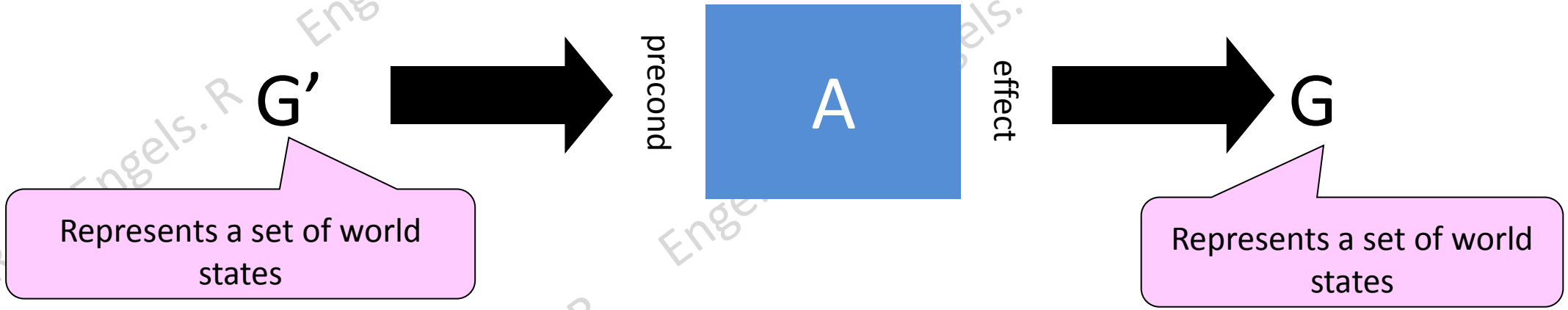
4. Removing Negative Effects (Only)

- 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 ORDERING GRAPHS – PLANNING GRAPHS

Regression

- Regressing a goal, G, thru an action, A
- Yields the weakest precondition G'
  - Such that: if G' is true before A is executed
  - G is guaranteed to be true afterwards



Planning Graphs

- All of the heuristics can suffer from inaccuracies
- A special data structure called a **Planning Graph** can be
  - used to give better heuristic estimates
  - applied to any of the search techniques or another algorithm GRAPHPLAN
- A planning problem asks ***if we can reach a goal state from the initial state***
- A tree of all possible actions from initial state to successor states, and their successors, and so on to answer
  - Can we reach state G from state S_0
  - This tree is of exponential size and hence impractical

Planning Graphs

- A planning graph = polynomial size approximation to the exhaustively indexed tree
  - can be constructed quickly
  - May not answer definitively whether G is reachable from S_0, but can estimate the number of steps
- The estimate is always correct when it reports the goal is not reachable
- And it never overestimates the number of steps
- So it is an admissible heuristic

Planning Graphs - 2

- Planning graph is a **directed graph** organized into **levels**
  - First a level S_0 for the initial state, consisting of nodes representing each fluent (a condition that can change over time) that holds in S_0
  - then a level A_0, consisting of nodes for each ground action that might be applicable in S_0
  - then alternating levels Si followed by Ai
  - until we reach a termination condition
- Planning graphs work *only for propositional planning problems*
  - The problems with no variables

Planning Graphs - 3

- Roughly
  - S_i contains all the literals that *could* hold at time I
    - depending on the actions executed at preceding time steps
  - If it is possible that either P or ¬P could hold, then both will be represented in S_i
  - A_i contains all the actions that *could* have their preconditions satisfied at time I
- Why roughly?
  - Planning graph records only a restricted subset of the possible negative interactions among actions;
  - therefore, a literal might show up at level Sj when actually it could not be true until a later level, if at all
  - It is guaranteed thought that A literal will never show up too late
- Despite the possible error, ***the level j at which a literal first appears is a good estimate of how difficult it is to achieve the literal from the initial state***

Planning Graphs – Example problem

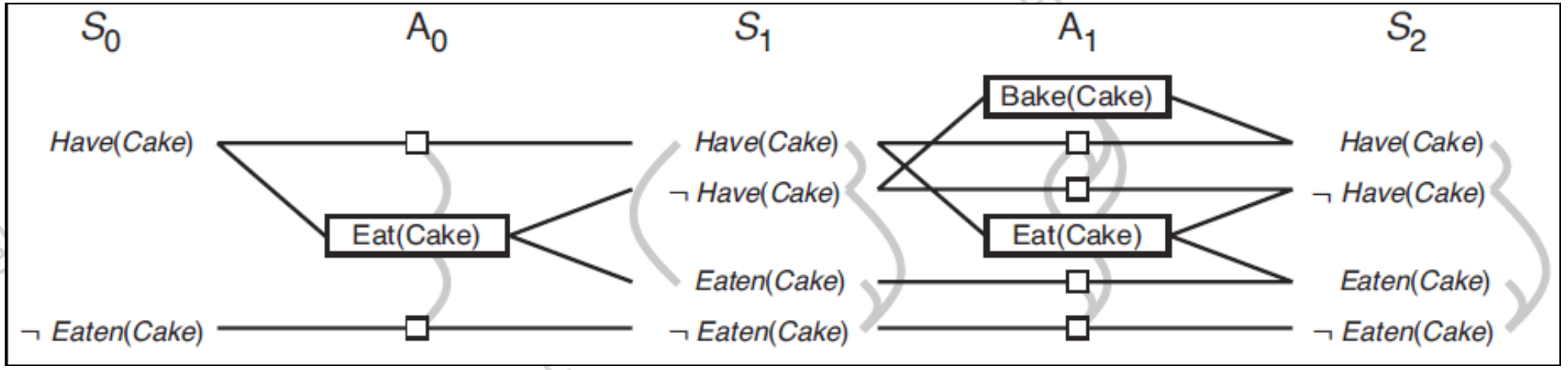
$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg Have(Cake)$
EFFECT: $Have(Cake))$

Figure 10.7 The "have cake and eat cake too" problem.

Planning Graphs – Example

- 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
- **Straight lines** indicate preconditions and effects
- **Mutex links** are shown as curved gray lines
  - (not all mutex links are shown)
- If two literals are mutex at Si, then the persistence actions for those literals will be mutex at Ai
  - need not draw that mutex link

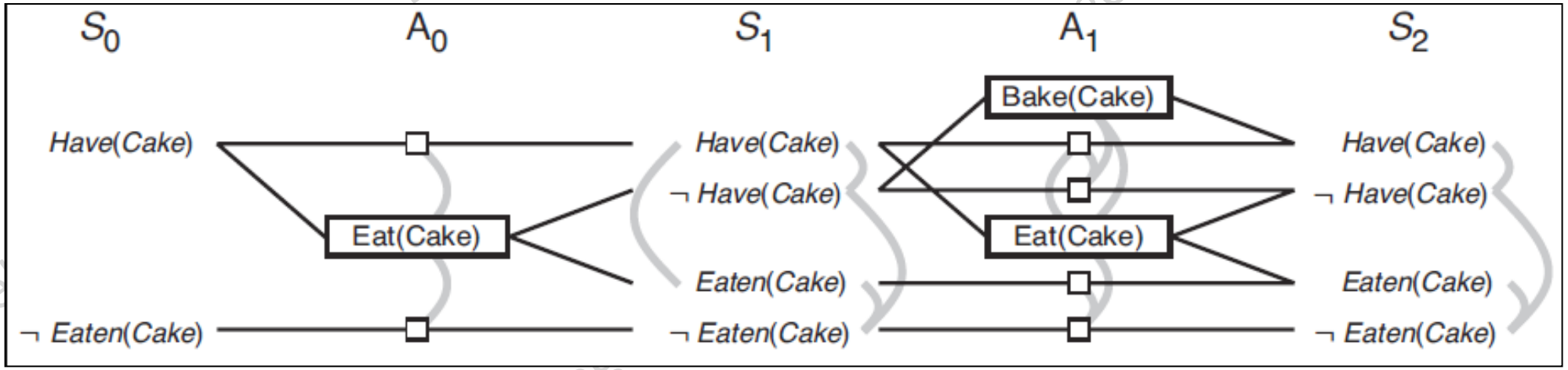
$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg\, Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg\, Have(Cake)$
EFFECT: $Have(Cake))$



Planning Graphs – Example

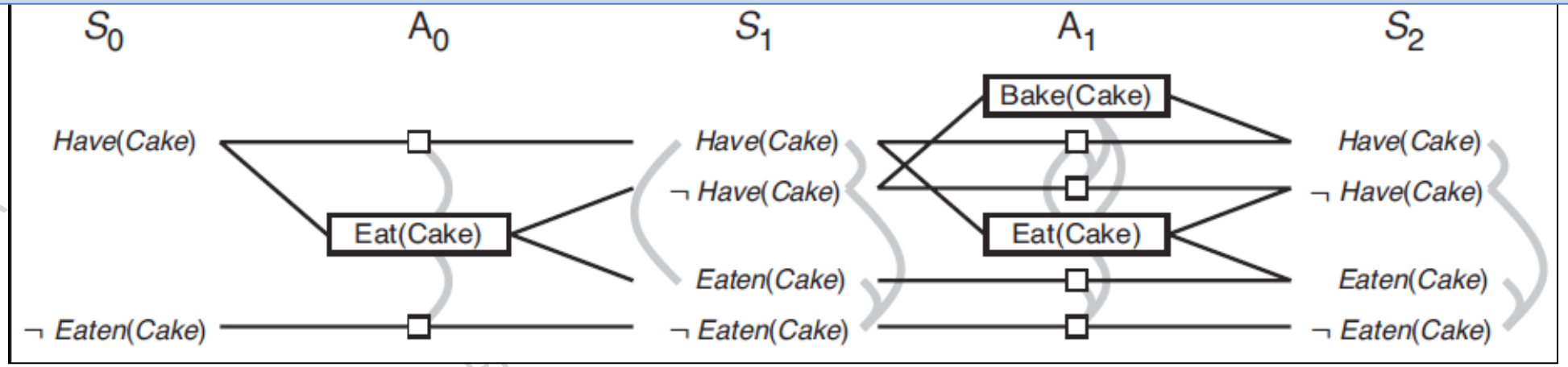
- Each action at level Ai is connected to its preconditions at Si and its effects at Si+1
- So a literal appears because an action caused it AND a literal can persist if no action negates it
  - Known as **persistence action** or **no-op**
- For every literal C, add to the problem a persistence action with precondition C and effect C
- Level A_0 in figure shows one "real" action, ***Eat (Cake)***
  - along with two persistence actions drawn as small square boxes

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



Planning Graphs – Example

- Level A_0 contains all the actions that ***could*** occur in state S_0
- Also records conflicts between actions that would prevent them from occurring together
- Grey links indicate **mutual exclusion** (or **mutex**) links
- For example, Eat (Cake) is mutually exclusive with the persistence of either Have(Cake) or ¬Eaten(Cake)
  - Also Have(Cake) and Eaten(Cake) are mutex
- S_1 represents **a belief state**: a set of ***possible*** states
- Continue alternating between state level Si and action level Ai until two consecutive levels are identical
  - Graph has **leveled off**
- Planning graph **does *not* require** choosing among actions
  - which would entail combinatorial search
- It **just records the impossibility of certain choices *using mutex links***



Mutex Relation

- Mutex relation holds between two ***actions*** at a given level if any of the following three conditions holds:
- ***Inconsistent effects:***
  - One action negates an effect of the other
  - Ex: **Eat (Cake)** and the persistence of **Have(Cake)** have inconsistent effects because they disagree on the effect **Have(Cake)**
- ***Interference:***
  - one of the effects of one action is the negation of a precondition of the other
  - Ex: **Eat (Cake)** interferes with the persistence of **Have(Cake)** by negating its precondition
- ***Competing needs:***
  - One of the preconditions of one action is mutually exclusive with a precondition of the other
  - Ex: **Bake(Cake)** and **Eat (Cake)** are mutex because they compete on the value of the **Have(Cake)** precondition
- ***Inconsistent support***
  - A mutex relation holds between two literals at the same level if one is the negation of the other
  - or if each possible pair of actions that could achieve the two literals is mutually exclusive

Planning Graph complexity

- A planning graph is polynomial in the size of the planning problem
- For a planning problem with **l** literals and **a** actions
  - each $\mathbf{S_i}$ has no more than **l** nodes and $\mathbf{l^2}$ mutex links, and
  - each $\mathbf{A_i}$ has no more than **a + l** nodes (including the no-ops), $\mathbf{(a + l)^2}$ mutex links, and 2(al + l) precondition and effect links
- Thus, an entire graph with **n** levels has a size of $\mathbf{O(n(a + l)^2)}$
- The time to build the graph has the same complexity

The GRAPHPLAN algorithm

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

    graph ← INITIAL-PLANNING-GRAPH(problem)
    goals ← CONJUNCTS(problem.GOAL)
    nogoods ← an empty hash table
    for tl = 0 to ∞ do
        if goals all non-mutex in S_t of graph then
            solution ← EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
            if solution ≠ failure then return solution
        if graph and nogoods have both leveled off then return failure
        graph ← EXPAND-GRAPH(graph, problem)
```

Figure 10.9 The GRAPHPLAN algorithm. GRAPHPLAN calls EXPAND-GRAPH to add a level until either a solution is found by EXTRACT-SOLUTION, or no solution is possible.

- The **GRAPHPLAN** algorithm **repeatedly adds a level** to a planning graph with **EXPAND-GRAPH**
- Once all the goals show up as nonmutex in the graph, **GRAPHPLAN** calls **EXTRACT-SOLUTION** to **search for a plan** that solves the problem
- If that fails, it expands another level and tries again, terminating with failure when there is no reason to go on

Planning Graphs – Spare tire example

$$
\begin{aligned}
&Init(Tire(Flat) \wedge Tire(Spare) \wedge At(Flat, Axle) \wedge At(Spare, Trunk)) \\
&Goal(At(Spare, Axle)) \\
&Action(Remove(obj, loc), \\
&\quad \text{PRECOND: } At(obj, loc) \\
&\quad \text{EFFECT: } \neg At(obj, loc) \wedge At(obj, Ground)) \\
&Action(PutOn(t, Axle), \\
&\quad \text{PRECOND: } Tire(t) \wedge At(t, Ground) \wedge \neg At(Flat, Axle) \\
&\quad \text{EFFECT: } \neg At(t, Ground) \wedge At(t, Axle)) \\
&Action(LeaveOvernight, \\
&\quad \text{PRECOND:} \\
&\quad \text{EFFECT: } \neg At(Spare, Ground) \wedge \neg At(Spare, Axle) \wedge \neg At(Spare, Trunk) \\
&\qquad \wedge \neg At(Flat, Ground) \wedge \neg At(Flat, Axle) \wedge \neg At(Flat, Trunk))
\end{aligned}
$$

Planning Graphs – Spare tire example

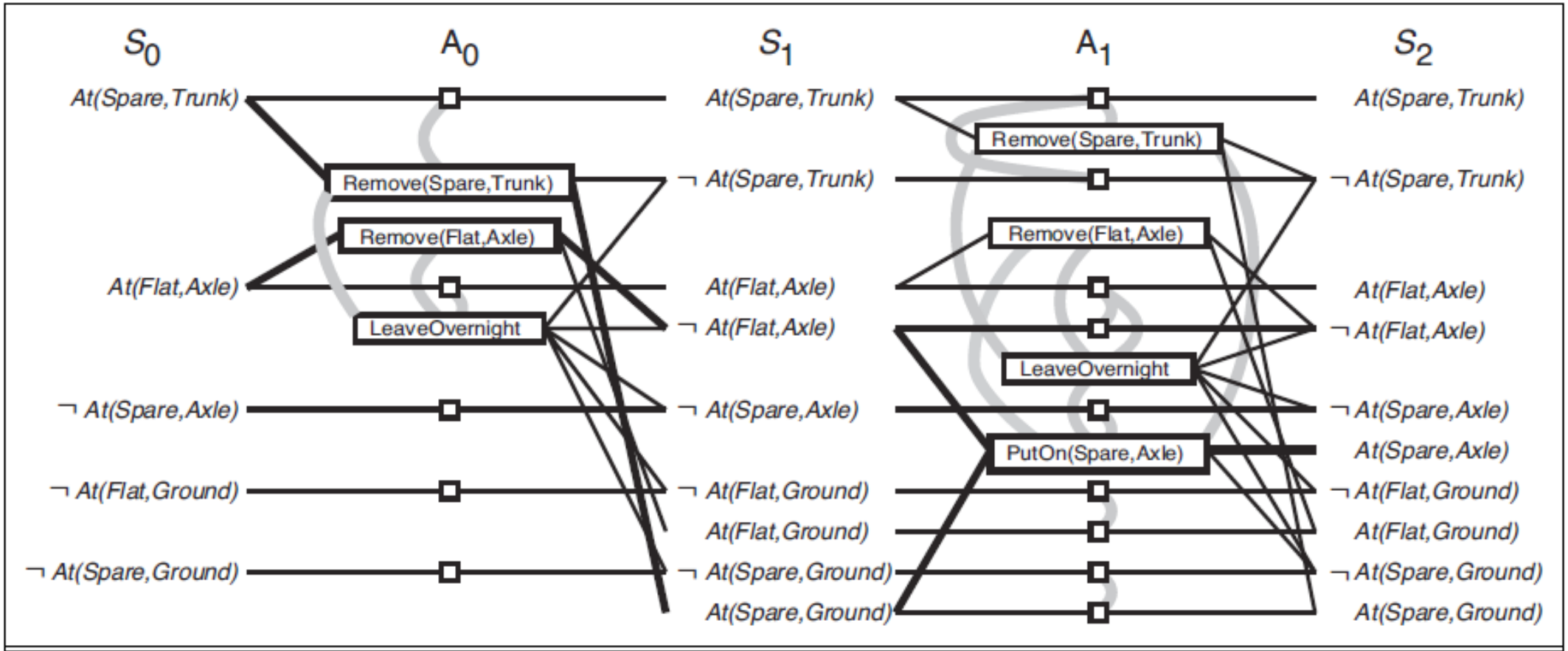


Figure 10.10 The planning graph for the spare tire problem after expansion to level S_2. Mutex links are shown as gray lines. Not all links are shown, because the graph would be too cluttered if we showed them all. The solution is indicated by bold lines and outlines.

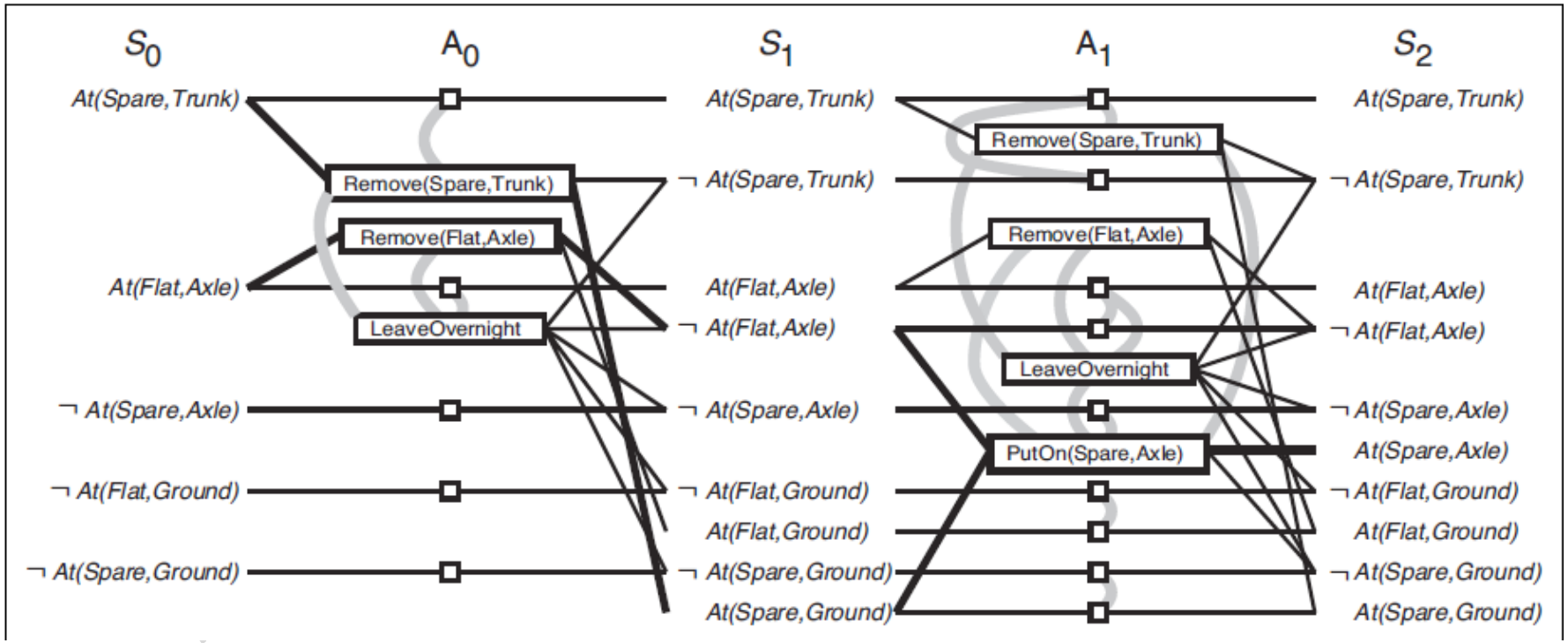
Planning Graphs – Spare tire example

- GRAPHPLAN **initializes** the planning graph to a one-level (S_0) graph representing the initial state
  - The positive fluents and the relevant negative fluents from the problem description's initial state are shown
  - Not shown are the unchanging positive literals (such as Tire(Spare)) and the irrelevant negative literals
- The goal **At(Spare, Axle)** is not present in S_0, so we need not call EXTRACT-SOLUTION
  - There is no solution yet
- EXPAND-GRAPH adds into A_0 the three actions whose preconditions exist at S_0
  - all the actions except PutOn(Spare, Axle)
  - along with persistence actions for all the literals in S_0
- The effects of the actions are added at level S_1
- EXPAND-GRAPH then looks for mutex relations and adds them to the graph
- At(Spare, Axle) is still not present in S_1
  - so again EXTRACT-SOLUTION is not called
- EXPAND-GRAPH is called again, adding A_1 and S_2 resulting in the planning graph shown

Planning Graphs – Spare tire example – Mutex examples

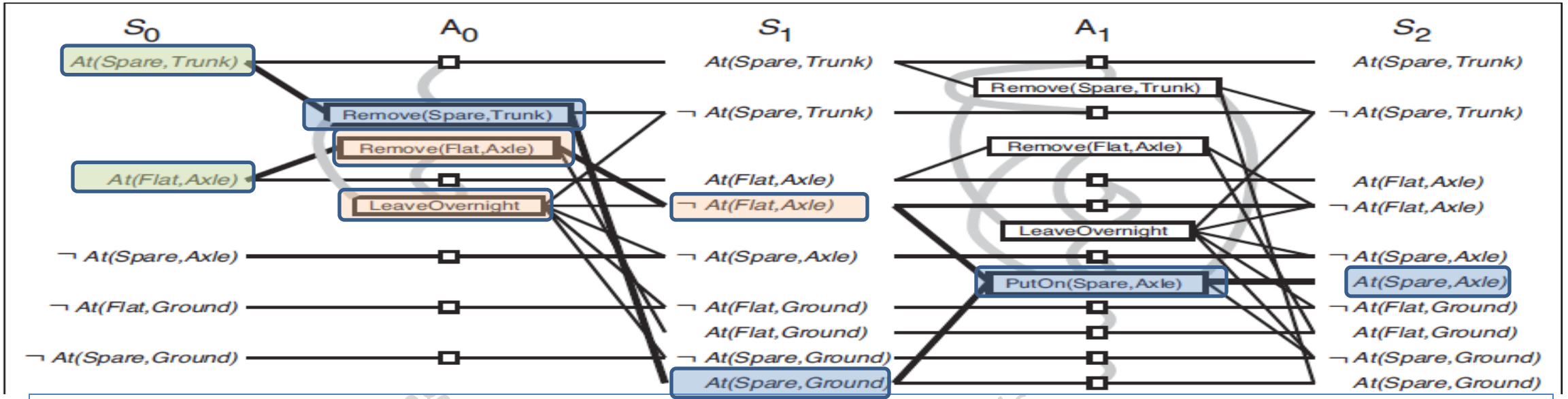
- ***Inconsistent effects:***
  - **Remove(Spare, Trunk)** is mutex with **LeaveOvernight** because one has the effect **At(Spare, Ground)** and the other has its negation
- ***Interference:***
  - **Remove(Flat, Axle)** is mutex with **LeaveOvernight** because one has the precondition **At(Flat, Axle)** and the other has its negation as an effect.
- ***Competing needs:***
  - **PutOn(Spare, Axle)** is mutex with **Remove(Flat, Axle)** because one has **At(Flat, Axle)** as a precondition and the other has its negation
- ***Inconsistent support:***
  - **At(Spare, Axle)** is mutex with **At(Flat, Axle)** in S_2
  - because the ***only*** way of achieving **At(Spare, Axle)** is by **PutOn(Spare, Axle)**
  - and that is mutex with the persistence action that is the **only way** of achieving **At(Flat, Axle)**
- Thus, the ***mutex relations detect the immediate conflict***
  - that arises from trying to put two objects in the same place at the same time

Planning Graphs – Spare tire example



- All the literals from the goal are present in S_2, and none of them is mutex with any other
  - Hence a ***solution might exist,*** and EXTRACT-SOLUTION will try to find it
- **Can formulate EXTRACT-SOLUTION** as a Boolean constraint satisfaction problem (CSP)
- Or as a **as a backward search problem**, where
  - each state in the search contains a pointer to a level in the planning graph and a set of unsatisfied goals

Planning Graphs – Spare tire example



- Start at S_2 with the goal **At(Spare, Axle)**
- The **only choice** for achieving the goal set is **PutOn(Spare, Axle)**
- Can only take to a search state at S_1 with goals **At(Spare, Ground)** and **¬At(Flat, Axle)**
  - **At(Spare, Ground)** can be achieved only by **Remove(Spare, Trunk)**, and
  - **¬At(Flat, Axle)** can be achieved only 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)**
- Brings us to a search state at S_0 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 **A_0**, followed by **PutOn(Spare, Axle)** in **A_1**

Planning Graphs – Summary

- Planning is a search problem
  - Possible to apply search heuristics to improve the process
- One approach is to construct a “graph” of the search space
  - and use this to guide search
- This leads to the concept of a planning graph

Unit 4 – Planning - Summary

- **Planning and Learning:**
  - **Planning with State Space Search:**
    - Partial Order Planning
    - Planning Graphs
    - Examples
- Blended/Self learning
  - **Forms of Learning**
    - Inductive Learning
    - Explanation Based Learning
    - Statistical Learning
    - Learning With Complete Data

References

- AI
  - Artificial Intelligence – Elaine Rich, Kevin Knight, B. Nair 3rd Edition
- AIMA
  - Artificial Intelligence - A Modern Approach 3rd Edition - RUSSELL & NORVIG