



Classical Planning

CS4246/CS5446

AI Planning and Decision Making

Sem 1, AY2021-22



Topics

- Classical planning model (RN 11.1)
 - STRIPS Planning and PDDL
 - Factored plan representation
- Planning as search
 - Forward state-space search (RN 11.2.1)
 - Backward search (RN 11.2.2)
- SATPlan and Satisfiability (RN 11.2.3)
- Complexity of planning (RN 11, Bibliographical and Historical Notes)



Solving Planning Problems

- Planning Problem or Model
 - Appropriate abstraction of states, actions, effects, and goals (and costs and values)
- Planning Algorithm
 - Input: a problem
 - Output: a solution in the form of an action sequence
- Planning Solution
 - A **plan** or **path** from the initial state(s) to the goal state(s)
 - Any path; OR
 - An **optimal** path wrt to costs or values
 - A **goal state** that satisfies certain properties



A Bit of History

- Classical planning – still very useful today!
 - Also called STRIPS planning
- What is STRIPS?
 - STanford Research Institute Problem Solver
 - R. Fikes and N. Nilsson (1971). STRIPS: a new approach to the application of theorem proving to problem solving. Artificial Intelligence, 2:189-208
 - Included planning language for automated planning problems
- Video:
 - https://media.ed.ac.uk/media/1_uhxvxo4a



Classical Planning

- Definition:
 - Find a sequence of actions to achieve a goal in an environment that is:
 - Discrete
 - Deterministic
 - Static
 - Fully observable
- Main challenges:
 - How to **represent** the planning problem? – PDDL from STRIPS
 - How to **search** for a solution (plan)? – search and satisfiability
 - How to use **heuristics** to solve for a solution (plan) more efficiently?



Review: Search

- Uniform cost search
 - Breadth-first search
 - Depth-first search
 - Iterative-deepening search
- Heuristic search
 - Greedy best-first search
 - A* search
 - IDA* search
- Deriving heuristics
 - Admissible heuristics
 - Consistent heuristics
 - Relaxed problems



Review: Logical Notation

- **Term:**
 - An object in the world. Can be a variable, constant or function
- **Atomic sentence or Atom:**
 - A predicate symbol, optionally followed by a parenthesized list of terms
- **Literal:**
 - Atom or its negation
- **Ground literal:**
 - A literal with no variable
- **Sentence or Formula:**
 - Formed from atoms together with quantifiers (\forall , \exists), logical connectives (\wedge , \vee , \neg), equality symbol ($=$)
 - Sentence takes values True or False
- **Substitution:**
 - Replaces variables by terms
- **Unifier:**
 - Takes 2 sentences and returns a substitution that makes the sentence look identical, if one exists
 $Unify(Brother(John, x), Brother(y, James))$
 $= \{x = James; y = John\}$
 - For every pair of unifiable sentences, there is a **most general unifier (MGU)**, which is unique up to renaming and substitution of variables



Planning Model

PDDL and STRIPS Planning

Planning Domain Definition Language

- PDDL

- Derived from the original STRIPS planning language
- Logic-based, **lifted** propositional representation (to restricted FOL)
- Basic version for classical planning, extensions in active research

- Planning problem representation

	Examples
State	$At(P_1, SFO) \wedge At(P_2, SIN)$
Action	$Action(Fly(P_1, SFO, SIN))$ PRECOND: $At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(SIN)$ EFFECT: $\neg At(P_1, SFO) \wedge At(P_1, SIN)$
Goal	$At(p, SIN) \wedge Plane(p)$



Main Characteristics

- Factored representation – Why?
 - Represent state of the world as a set of **state variables** or **fluents**
 - Each fluent depicts an **aspect** of the world that changes with time
- Domain independent heuristics
 - Avoid need for domain specific heuristics to work well in planning
- On-demand state computation
 - “Implicit” transition function
 - Just specify the current state
 - Define actions that can compute state-transitions
 - Use actions to generate the other states as needed

State Representation

A set of fluents

- State
 - A conjunction of **ground atomic** fluents
 - Ground atomic – there is a single predicate, with only constant arguments; function-free
 - Examples:
 - $Hungry \wedge Sleepy$
 - $New(Plane_1) \wedge Safe(Plane_1)$
 - $At(Plane_1, SIN) \wedge At(Plane_2, SFO)$
- Database semantics:
 - **Closed-world** assumption – Any fluents that are not mentioned are False
 - e.g., $Fierce(CS4246_Lecturer)$
 - **Unique names** assumption – Any two objects with different names are different
 - e.g., $Plane_1$ and $Plane_2$



Exercises

- Are the following fluents allowed in a state?

$\neg Expensive$

$At(x, SR2)$

$At(BestFriend(WinnieThePooh), HundredAcreWoods)$



Goal Representation

- Goal State:
 - Goal is a partially specified state, represented as a conjunction of positive ground literals
 - A state s **satisfies** a goal g if s contains all the literals in g
 - Examples:
 - *Hungry \wedge Sleepy \wedge Bored* satisfies the goal *Hungry \wedge Bored*
 - *At(Cargo₁, SFO)* satisfies the goal *At(c , SFO)*
where c is a variable for any cargo

Action Schema

- Defines a family of ground actions
 - All variables assumed to be universally quantified
 - Lifts propositional logic to restricted subset of first-order logic
 - **Pre-Conditions:**
 - Conjunctions of literals (positive and negative atomic sentences) defining the applicable states in which the action can be executed
 - **Effects:**
 - Conjunctions of literals defining the result of executing the action

Action(Fly(p, from, to)

Precond: At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport (to)

Effect: \neg At(p, from) \wedge At(p, to))

Grounded Actions

- Grounded actions
 - When an action schema A contains variables, it may have multiple applicable **instantiations** or **grounded actions**

Action(Fly(P_1 , SFO, SIN))

PRECOND: $At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(SIN)$

EFFECT: $\neg At(P_1, SFO) \wedge At(P_1, SIN)$

Action(Fly(P_2 , SIN, SFO))

PRECOND: $At(P_2, SIN) \wedge Plane(P_2) \wedge Airport(SIN) \wedge Airport(SFO)$

EFFECT: $\neg At(P_2, SIN) \wedge At(P_2, SFO)$



Exercise

- Question:

- For an action a with v variables in a domain with k objects, how many distinct ground actions are there in the worst case?
- Example action: *Fly* (p ; *from*; *to*).

Action Execution

- Current state and available action:

$At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(SIN)$

$Action(Fly(P_1, SFO, SIN))$

Precond: $At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(SIN)$

Effect: $\neg At(P_1, SFO) \wedge At(P_1, SIN)$

- Execute action

$Fly(P_1, SFO, SIN)$

- Remove from database of states

$At(P_1, SFO)$

- Add to database of states

$At(P_1, SIN)$

Action Effects

- Applicable State

- A ground action a is **applicable** in state s if s **entails** the precondition of a

- Execution Result

- Result of executing applicable action a in state s is a state s' formed by:
 - Removing fluents in action effects that are negative literals - **delete list** $DEL(a)$, and
 - Adding fluents in action effects that are positive literals— **add list** $ADD(a)$

$$s' = (s - DEL(a)) \cup ADD(a)$$

- New states are generated **on demand** with the $ADD(a)$ and $DEL(a)$ functions as a result of execution an action

Exercise

- What is the result of executing the action

Fly (P_1 , *SFO*, *SIN*) from the following state?

At(P_1 ; *SFO*) \wedge *Plane*(P_1) \wedge *Airport*(*SFO*) \wedge *Airport*(*SIN*)

Action(*Fly*(p ; *from*; *to*);

Precond : *At*(p ; *from*) \wedge *Plane*(p) \wedge *Airport*(*from*) \wedge *Airport*(*to*)

Effect : *At*(p ; *from*) \wedge *At*(p ; *to*)



The Frame Problem

- Definition
 - What changes and what stays as the **result of the action**
 - Needs to be addressed by any real-world planning models
- Action representation in PDDL:
 - Times and states are **implicit** in action schemas:
 - Preconditions always refers to time t and the effect to time $t + 1$
 - Specifies result of an action in terms of changes; everything that stays the same unmentioned
- State representation in PDDL:
 - Uses **closed world assumption** that anything not mentioned is FALSE
 - The fluents do not explicitly refer to time

Planning Problem and Solution

- Initial state

- A state – a conjunction of ground atoms
- Example: $A \wedge t(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(SIN) \wedge \dots$

- Goal

- A conjunction of literals that may contain variables
- Any variables are treated as **existentially quantified**
- Example: $At(p, SIN) \wedge Plane(p)$.

STRIPS planning restricts
Goal to positive literals
with no variables

- Problem solution

- A sequence of actions that end in a state s that entails the goal
- Example: Any plan that reaches state:

$Plane(P_1) \wedge At(P_1, SIN)$ entails goal $At(p, SIN) \wedge Plane(p)$

Example: Air Cargo Transport Planning

Init($At(C_1, SIN) \wedge At(C_2, SIN) \wedge At(P_1, SFO) \wedge At(P_2, SIN)$
 $\wedge Cargo(C_1) \wedge Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2)$
 $\wedge Airport(SIN) \wedge Airport(SFO)$)
Goal($At(C_1, SIN) \wedge At(C_2, SFO)$)

Action(*Load*(c, p, a),
PRECOND: $At(c, a) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$
EFFECT: $\neg At(c, a) \wedge In(c, p)$)
Action(*Unload*(c, p, a),
PRECOND: $In(c, p) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$
EFFECT: $At(c, a) \wedge \neg In(c, p)$)
Action(*Fly*($p, from, to$),
PRECOND: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
EFFECT: $\neg At(p, from) \wedge At(p, to)$)

Planning Domain

Planning Problem

Adapted from source: RN Figure 11.1



A Possible Plan

- A solution –
- Some caveats
 - What to do with spurious actions like *Fly*(P_1 , *SIN*, *SIN*)?

Planning in PDDL

- Planning domain: A set of action schemas
- Planning problem: Planning domain + initial state + goal
- Planning model summary
 - State: conjunction of ground positive function-free literals
Closed world assumption
 - Goal: conjunction of (ground positive) function-free literals
Satisfied by states that contain all literals
 - Action precondition: conjunction of function-free literals
 - Action effects: conjunction of function-free literals
 - Positive literals for add list
 - Negative literals for delete list

STRIPS
Planning



Algorithms for Classical Planning

- Possible approaches
 - **Forward search** through state space from initial state to goal
 - **Backward search** through state space from goal to initial state
 - Translate problem description into set of logic sentences and apply **logical inference** algorithm to find solution (**SATPlan**)



Planning as State-Space Search

Forward (progression) search

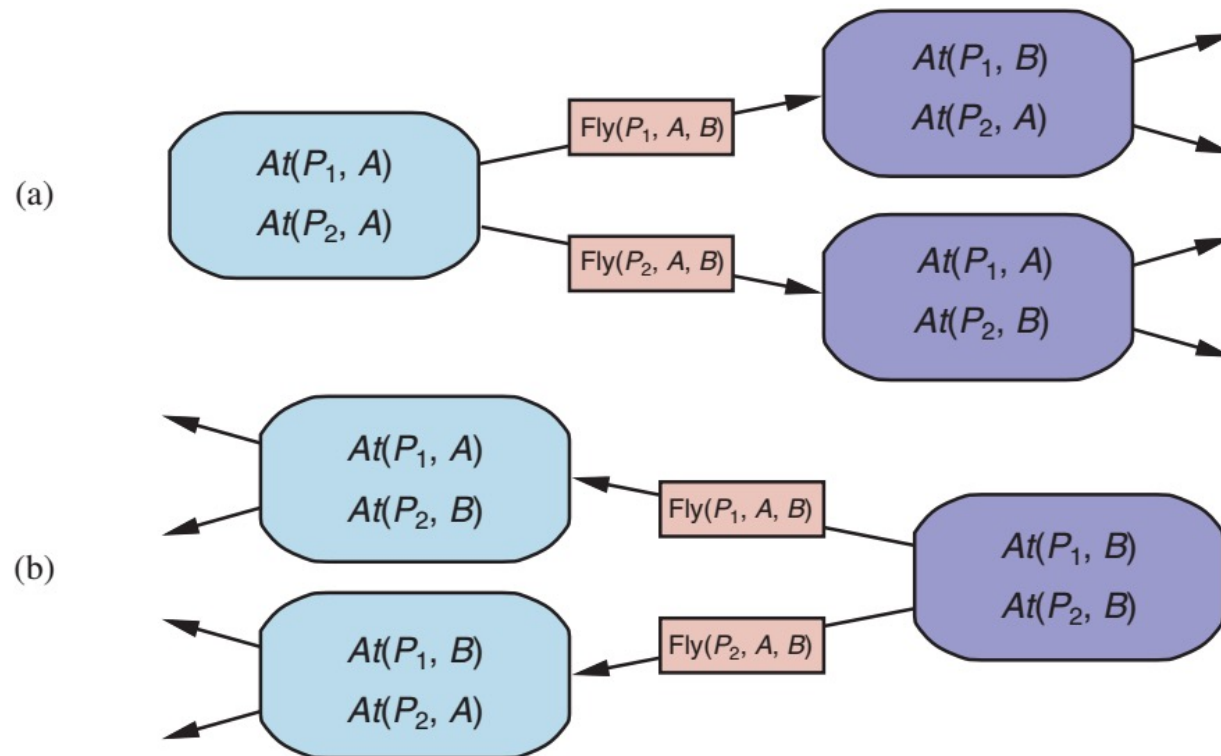
Backward (regression) search



Planning as State-Space Search

- Map planning problem into **search problem** in state-space
 - States are ground states with only binary fluents
 - Goal state has all the positive fluents in problem's goal
 - Applicable actions in a state are ground actions
- Planning search tree :
 - Each **node** is a state, including initial state and goal state
 - Each **branch** is an action that allows a transition from one state to another
 - A **plan** is a **path from the initial state to the goal state in search tree**
- Algorithms:
 - Forward (progression) state-space search
 - Backward (regression) relevant-states search

Forward and Backward Search



Source: RN Figure 11.5

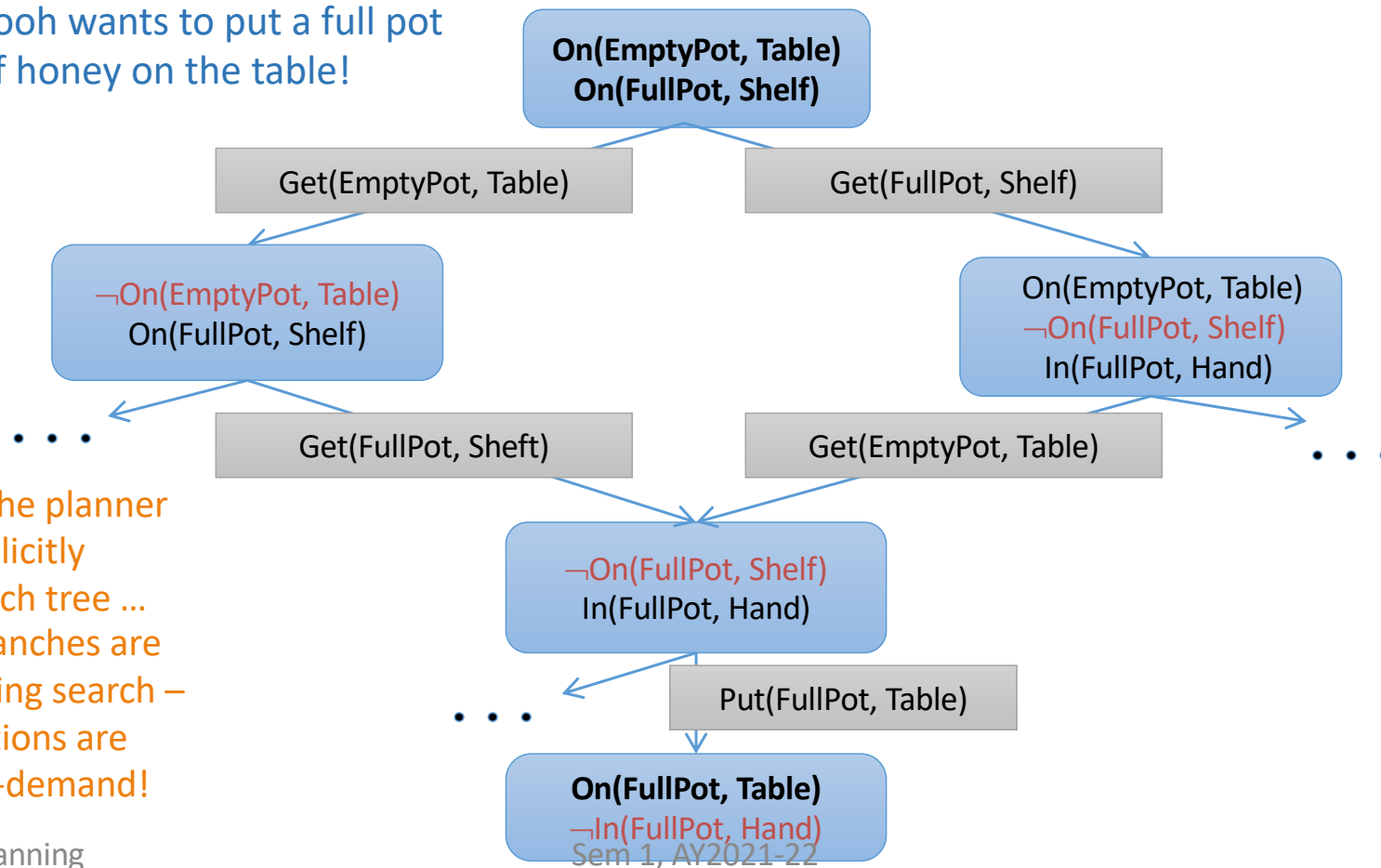


Example: Honey on Table

- Initial state
 - $\text{On}(\text{EmptyPot}, \text{Table}) \wedge \text{On}(\text{FullPot}, \text{Shelf})$
 - $\text{Pot}(\text{FullPot}) \wedge \text{Pot}(\text{EmptyPot}) \wedge \text{Place}(\text{Shelf}) \wedge \text{Place}(\text{Table})$
- Goal
 - $\text{On}(\text{FullPot}, \text{Table})$
- Actions
 - **Get**(FullPot, place)
 - Precond: $\text{On}(\text{FullPot}, \text{place})$
 - Effect: $\neg \text{On}(\text{FullPot}, \text{place}) \wedge \text{In}(\text{FullPot}, \text{Hand})$
 - **Get**(EmptyPot, place)
 - Precond: $\text{On}(\text{EmptyPot}, \text{place})$
 - Effect: $\neg \text{On}(\text{EmptyPot}, \text{place})$ - This goes straight to the Trash, don't hold in Hand
 - **Put**(pot, place)
 - Precond: $\neg \text{On}(\text{pot}, \text{place}) \wedge \text{In}(\text{pot}, \text{Hand})$
 - Effect: $\text{On}(\text{pot}, \text{place}) \wedge \neg \text{In}(\text{pot}, \text{Hand})$

Example: State-Space Representation

Pooh wants to put a full pot of honey on the table!



Remember: The planner does NOT explicitly build the search tree ... nodes and branches are traversed during search – states and actions are computed on-demand!

Planning by Forward Search

Start at a state s

Repeat:

1. Stop if goal g is satisfied
 - Check if the goal is a subset of the state s description: $g \subseteq s$
2. Compute applicable actions
 - Check if pre-conditions of an action a is a subset of the state s description:
 $Precond(a) \subseteq s$
3. Compute successor states
 - Add the list of positive effects of an action a to the state s description;
delete the list of negative effects: $s' = (s - DEL(a)) \cup ADD(a)$
4. Pick a successor state s' as current state



Example: Forward Search

Start at initial state
Is goal satisfied?

On(EmptyPot, Table)
On(FullPot, Shelf)

Goal: On(FullPot, Table)

Example: Forward Search

Compute Applicable Actions

On(EmptyPot, Table)
On(FullPot, Shelf)

Goal: On(FullPot, Table)

Get(EmptyPot, Table)

Get(FullPot, Shelf)

Get(EmptyPot, place)

Precond: On(EmptyPot, place)

Effect: \neg On(EmptyPot, place)

Subst = {place/Table}

Get(FullPot, place)

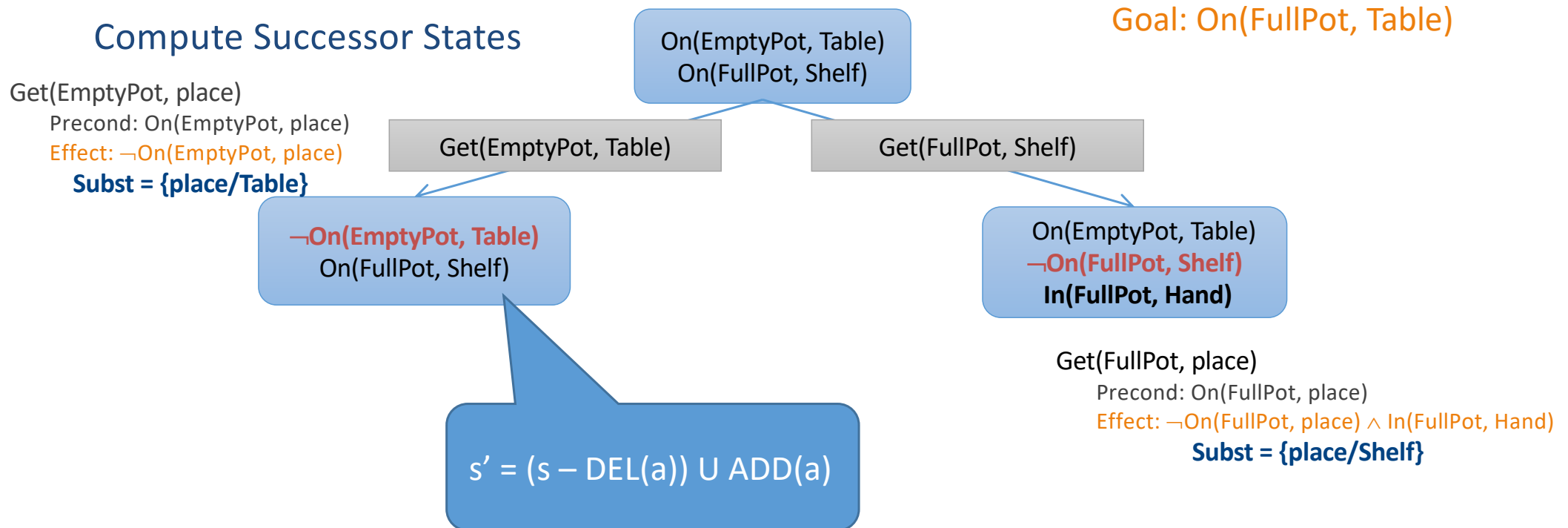
Precond: On(FullPot, place)

Effect: \neg On(FullPot, place) \wedge In(FullPot, Hand)

Subst = {place/Shelf}

Example: Forward Search

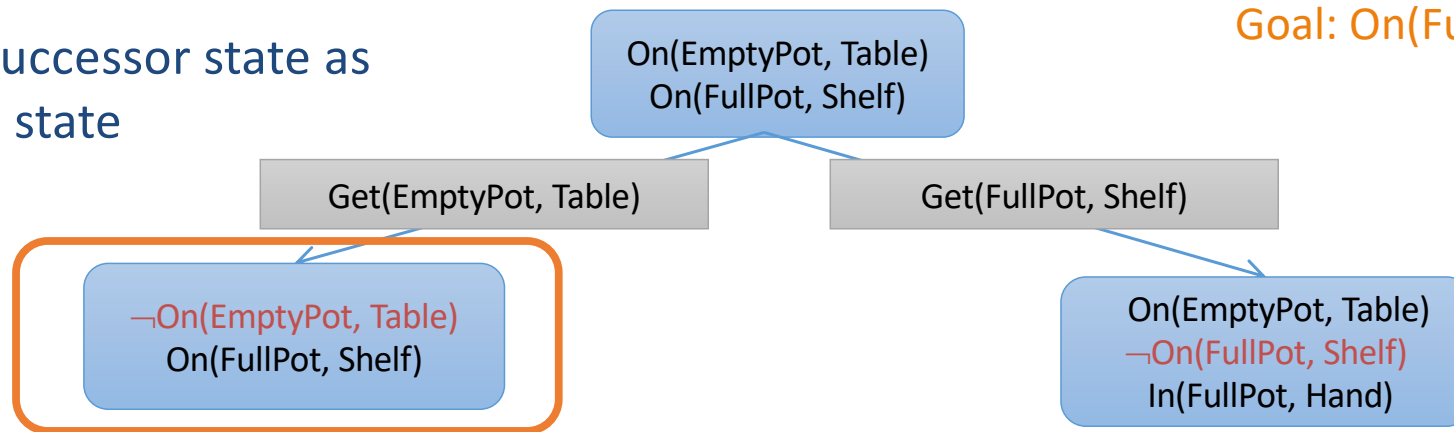
Compute Successor States



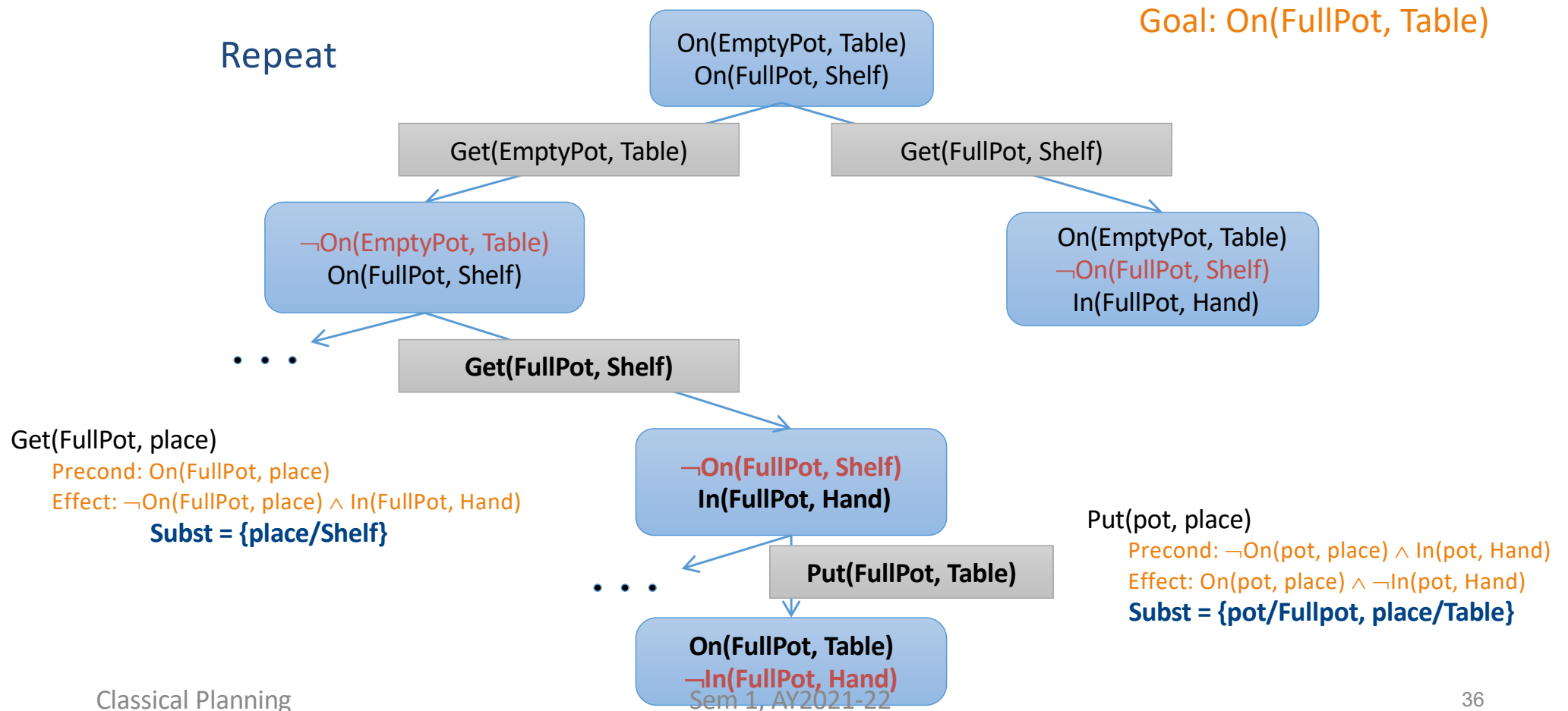
Example: Forward Search

Pick a successor state as current state

Goal: On(FullPot, Table)



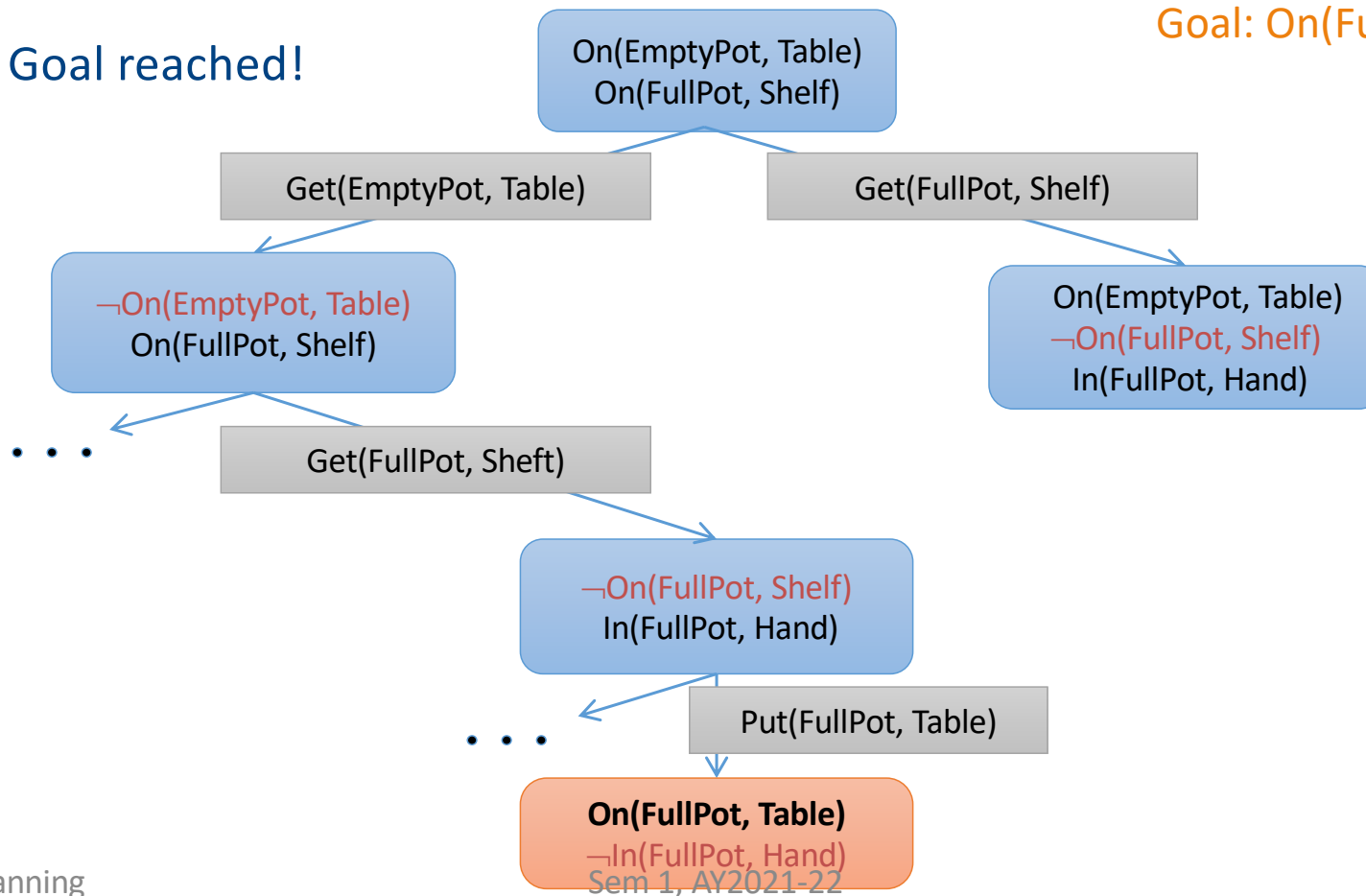
Example: Forward Search



Example: Forward Search

Success! Goal reached!

Goal: $\text{On}(\text{FullPot}, \text{Table})$

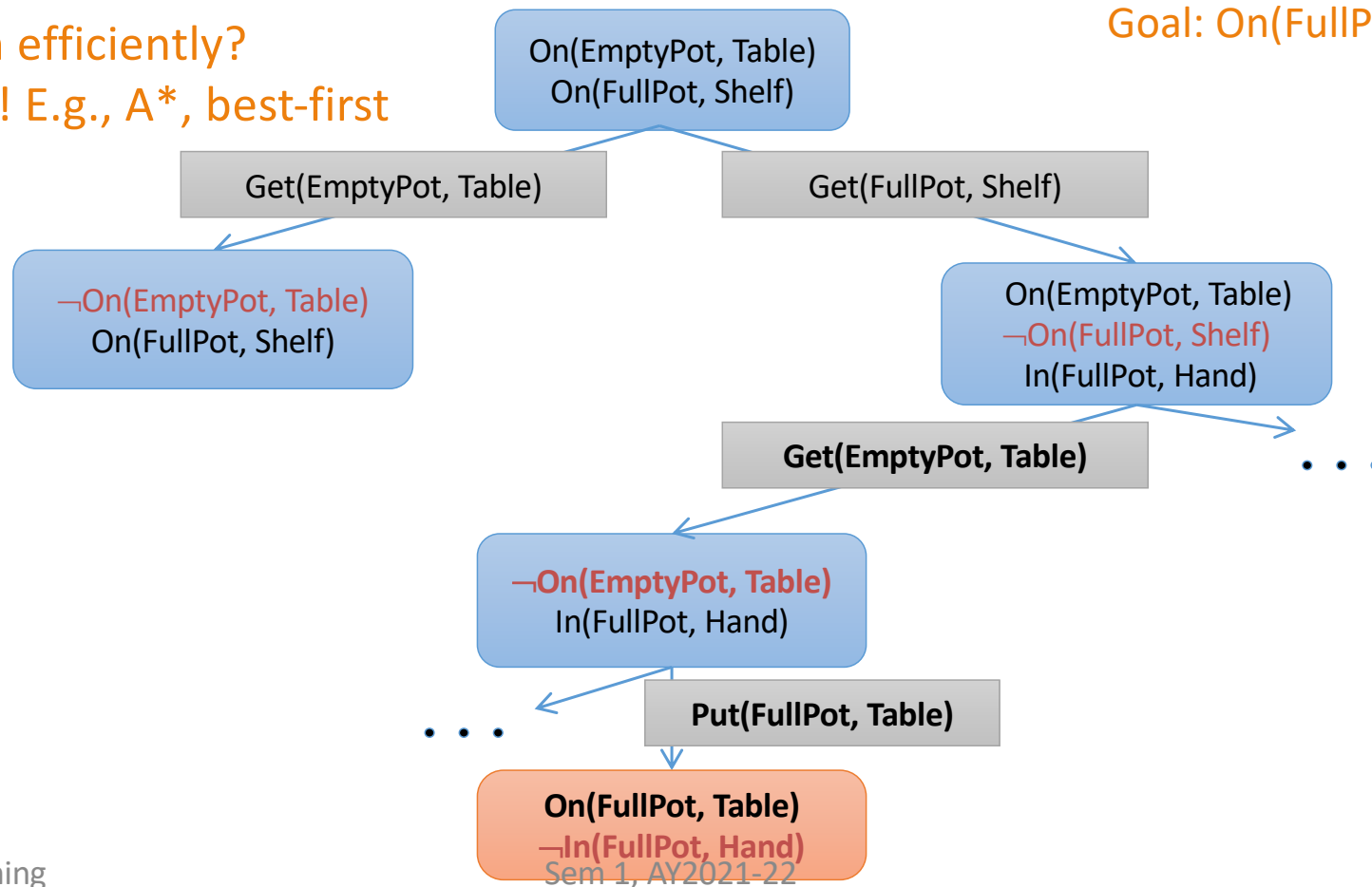


Example: Forward Search – Alternate Path

How to search efficiently?

Use heuristics! E.g., A*, best-first

Goal: On(FullPot, Table)



Planning by Backward Search

- Start at goal g

Repeat:

1. Stop if initial state s is satisfied

- Check if the initial state s is a subset of the goal state g description: $s \subseteq g$

2. Compute relevant actions

- Check if effects of an action a is a subset of the goal state g description: $Effects(a) \subseteq g$

3. Compute predecessor states

- Add the preconditions and remove the list of positive and negative effects of an action a to the current goal state g description

$$POS(g') = (POS(g) - ADD(a)) \cup POS(Precond(a))$$

$$NEG(g') = (NEG(g) - DEL(a)) \cup NEG(Precond(a))$$

4. Pick a predecessor state g' to be current goal state



Example: Backward Search

Start at goal state

Is initial state satisfied?

Initial state:

$\text{On}(\text{EmptyPot}, \text{Table}) \wedge \text{On}(\text{FullPot}, \text{Shelf})$

$\text{On}(\text{FullPot}, \text{Table})$

$\neg \text{In}(\text{FullPot}, \text{Hand})$

Sem 1, AY2021-22

Example: Backward Search

Compute relevant actions

Initial state:

$\text{On}(\text{EmptyPot}, \text{Table}) \wedge \text{On}(\text{FullPot}, \text{Shelf})$

Put(FullPot, Table)

On(FullPot, Table)

$\neg \text{In}(\text{FullPot}, \text{Hand})$

Sem 1, AY2021-22

Put(pot, place)

Precond: $\neg \text{On}(\text{pot}, \text{place}) \wedge \text{In}(\text{pot}, \text{Hand})$

Effect: $\text{On}(\text{pot}, \text{place}) \wedge \neg \text{In}(\text{pot}, \text{Hand})$

Subst = {pot/Fullpot, place/Table}

Example: Backward Search

Compute predecessor states

Initial state:

$\text{On}(\text{EmptyPot}, \text{Table}) \wedge \text{On}(\text{FullPot}, \text{Shelf})$

$\text{POS}(g') = (\text{POS}(g) - \text{ADD}(a)) \cup \text{POS}(\text{Precond}(a))$
 $\text{NEG}(g') = (\text{NEG}(g) - \text{DEL}(a)) \cup \text{NEG}(\text{Precond}(a))$

$\neg \text{On}(\text{FullPot}, \text{Table})$
 $\text{In}(\text{FullPot}, \text{Hand})$

$\text{Put}(\text{FullPot}, \text{Table})$

$\text{On}(\text{FullPot}, \text{Table})$
 $\neg \text{In}(\text{FullPot}, \text{Hand})$
Sem 1, AY2021-22

$\text{Put}(\text{pot}, \text{place})$

$\text{Precond: } \neg \text{On}(\text{pot}, \text{place}) \wedge \text{In}(\text{pot}, \text{Hand})$

$\text{Effect: } \text{On}(\text{pot}, \text{place}) \wedge \neg \text{In}(\text{pot}, \text{Hand})$

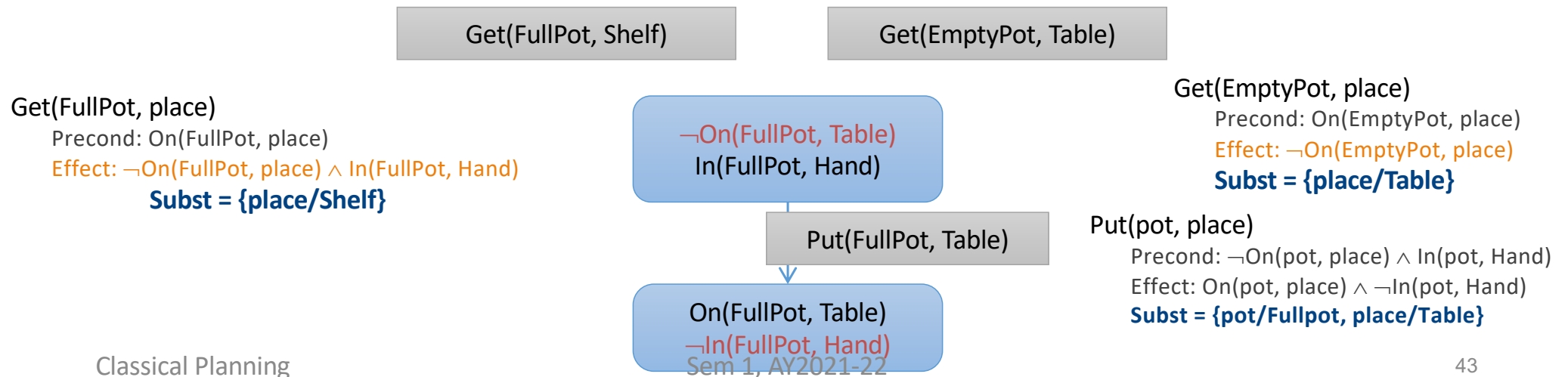
$\text{Subst} = \{\text{pot}/\text{Fullpot}, \text{place}/\text{Table}\}$

Example: Backward Search

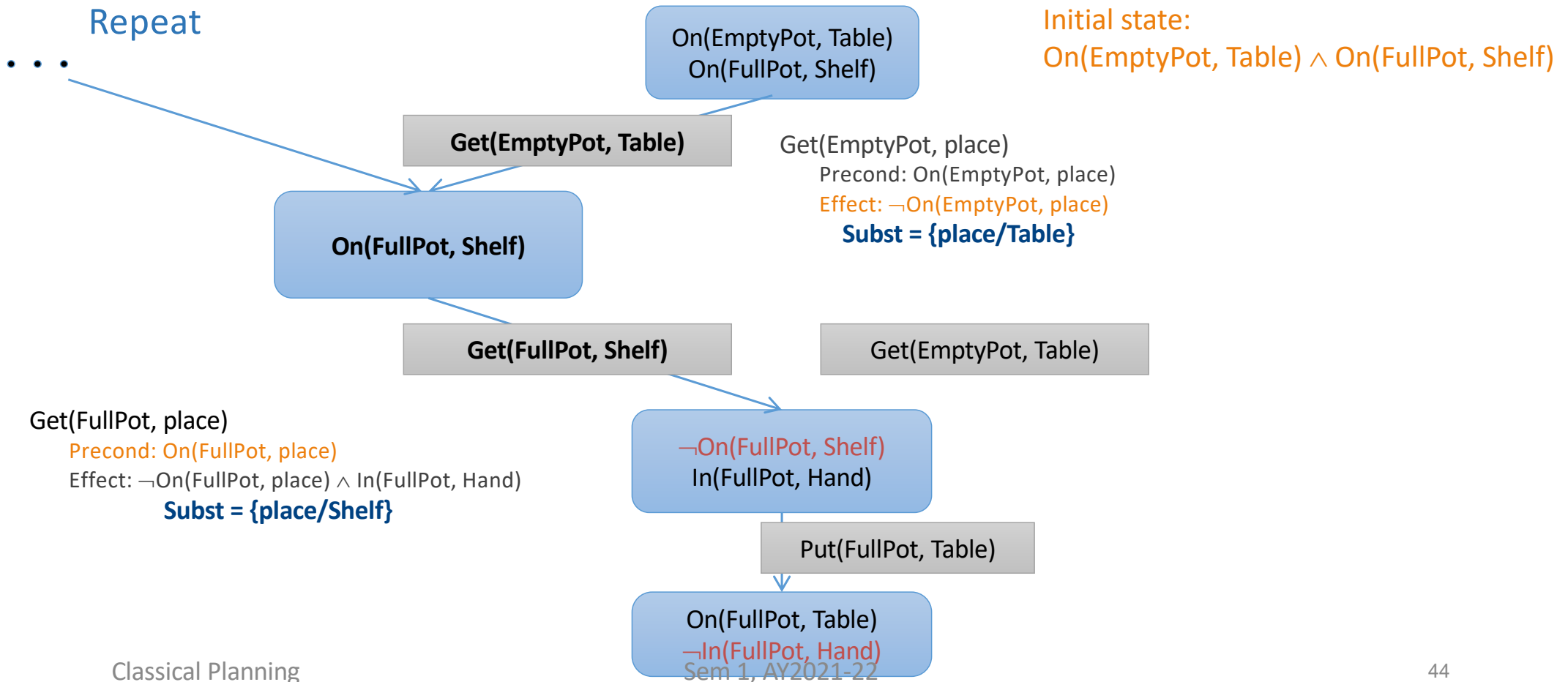
Pick a predecessor state
as the current goal state

Initial state:

$\text{On}(\text{EmptyPot}, \text{Table}) \wedge \text{On}(\text{FullPot}, \text{Shelf})$



Example: Backward Search

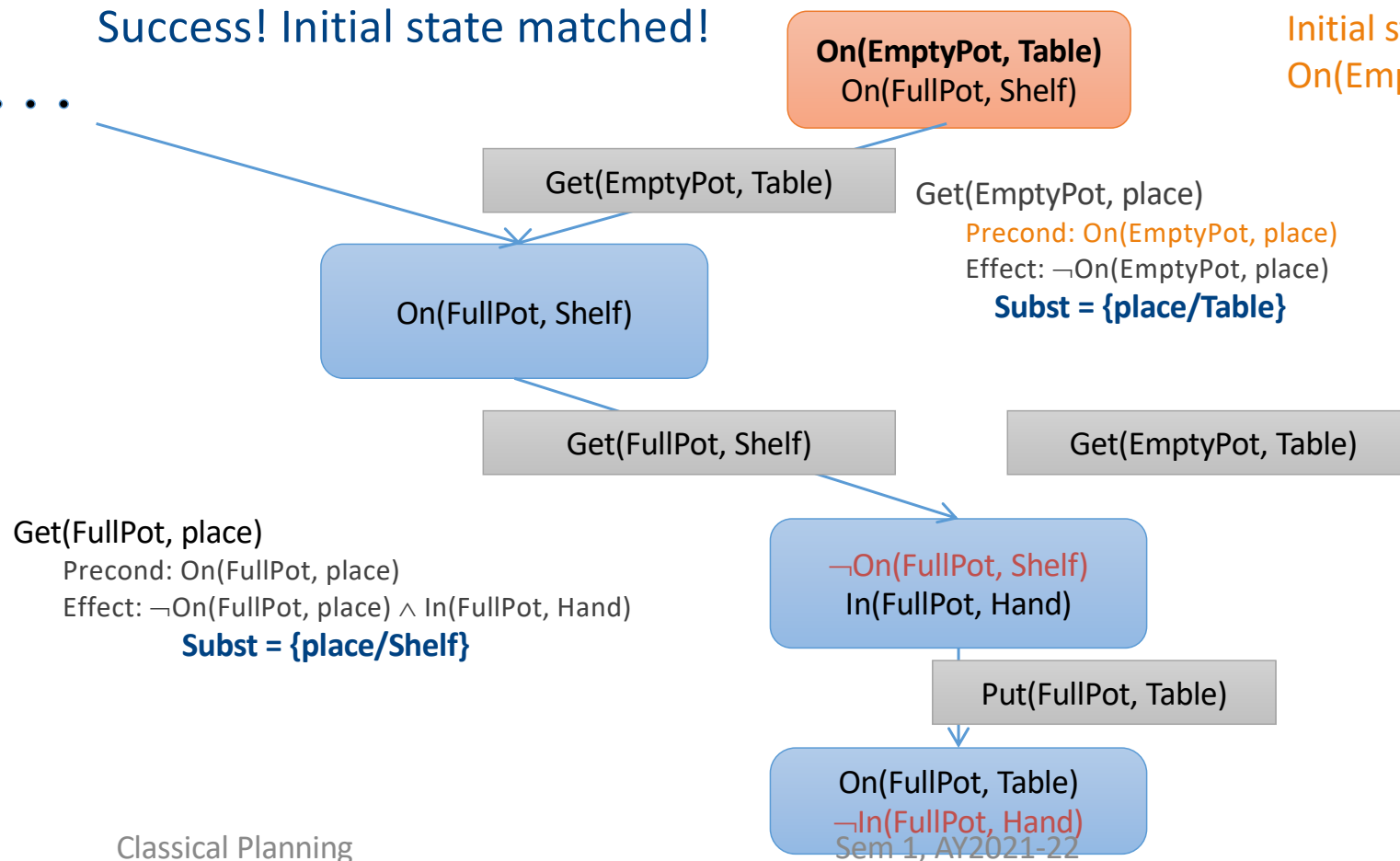


Example: Backward Search

Success! Initial state matched!

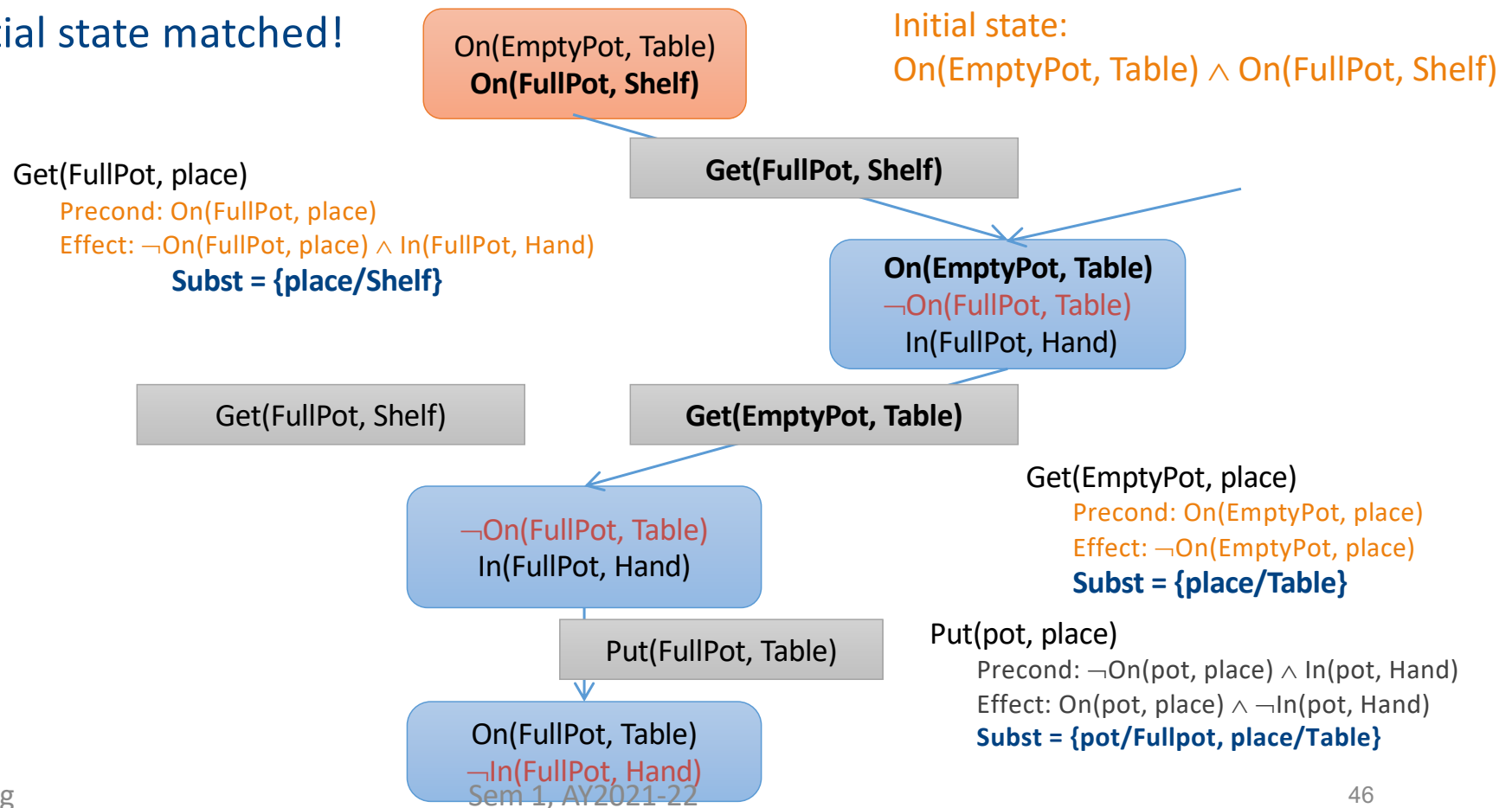
Initial state:

$\text{On}(\text{EmptyPot}, \text{Table}) \wedge \text{On}(\text{FullPot}, \text{Shelf})$



Example: Backward Search – Alternate Path

Success! Initial state matched!



State Descriptions in Regression

- Notes

- A set of **relevant** states to consider at each step (cf. forward search)
- Regression from a **state description** to a predecessor state description

- State vs Description

- In a state, every variable is assigned a value.
- For n ground fluents, there are 2^n ground states
- For n ground fluents, there are 3^n descriptions
 - Each fluent can be **positive, negative, or not mentioned**.
 - E.g. for goal: $\neg Hungry \wedge Sleepy$ describes states where *Hungry* is False and *Sleepy* is True, but other unmentioned fluents can have any value.
- A **description** represent a set of states.

Relevant Actions in Regression

- Relevant actions

- Those that can be the last step in a plan leading up to current goal state
- At least one of its effects must **unify** with an element of the goal;
- Must not have any effect that negates the elements of the goal
- Substitute **most general unifier** to keep branching factor down but not rule out any solution

- Regression

- Given a goal g and an action a , regression from g over a gives a state description g' whose positive and negative literals are given by:

$$\begin{aligned} Pos(g') &= (Pos(g) - Add(a) \cup Pos(Precond(a))) \\ Neg(g') &= (Neg(g) - Del(a) \cup Neg(Precond(a))) \end{aligned}$$

Relevant Actions in Regression

- More formally:

- Assume a goal description g that contains a goal literal g_i and an action schema A
- If A has an effect literal e'_j where $Unify(g_i, e'_j) = \theta$ and where we define $A' = Subst(\theta, A)$ and if there is no effect A' that is the negation of a literal in g , then A' is a relevant action towards g

Example

- Goal

- $At(C_2, SFO)$

- Relevant action schema

- $Action(Unload(c, p, a))$
- Precond: $In(c, p) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$
- Effect: $At(c, a) \wedge \neg In(c, p)$

- New action schema after substitution

- $Action(Unload(C_2, p', SFO))$
- Precond: $In(C_2, p') \wedge At(p', SFO) \wedge Cargo(C_2) \wedge Plane(p') \wedge Airport(SFO)$
- Effect: $At(C_2, SFO) \wedge \neg In(C_2, p')$

Substitution
 $\theta = \{c/C_2, a/SFO\}$

- Regressed state (new goal)

- $g' = In(C_2, p') \wedge At(p', SFO) \wedge Cargo(C_2) \wedge Plane(p') \wedge Airport(SFO)$

Example

- Question:
 - If the goal is $P \wedge Q \wedge R$
 - If Action a_1 has effect $P \wedge Q \wedge \neg R$, is a_1 a relevant action?
- Answer:



Forward and Backward Search in Large Domains

- Extended air cargo transportation problem
 - Air-cargo from airport A to airport B; 10 airports, each airport with 5 planes and 20 cargos
 - How to plan?
- Forward Search
 - Prone to exploring irrelevant actions
 - Need to traverse large state-spaces
 - Average branching factor is huge!
 - Need domain-specific or domain-independent heuristics that can be derived automatically
- Backward search
 - Keep branching factor lower than forward search
 - Using state sets for searching makes it harder to come up with heuristics



Planning as Boolean satisfiability

SATPlan

Planning as Boolean Satisfiability

- Transform classical planning problem into a Boolean satisfiability (SAT) problem
 - Translate planning problem into propositional formula in Conjunctive Normal Form (CNF)
 - Then determine whether propositional formula is **satisfiable**
 - Example:

$$(P \vee Q) \wedge (\neg Q \vee R \vee S) \wedge (\neg R \vee \neg P)$$

Does there exist a model i.e., an assignment of truth values to the proposition that makes the formula true?

- SAT is a NP-complete problem
 - SAT solvers are effective in practice; can solve problems with millions of variables and constraints
 - Use domain independent heuristics to search for a solution (See RN 7.7.4)



The SATPlan Algorithm

1. Construct a sentence that includes:
 - Init^0 – a collection of assertions about the initial state
 - $\text{Transition}^1, \dots, \text{Transition}^t$, the successor-state axioms for all possible actions at each time up to some maximum time t
 - Assertion that the goal is achieved at time t
2. Present whole sentence to SAT solver
 - If solver finds a satisfiable model, then goal is achievable
 - If sentence is unsatisfiable, then planning problem is impossible
3. Assume a model is found
 - Extract from model those variables that represent actions and are assigned true
 - Together they represent a plan to achieve the goals

The SATPlan Algorithm

function SATPLAN(*init*, *transition*, *goal*, T_{\max}) **returns** solution or *failure*
 inputs: *init*, *transition*, *goal*, constitute a description of the problem
 T_{\max} , an upper limit for plan length

for $t = 0$ **to** T_{\max} **do**
 $cnf \leftarrow$ TRANSLATE-TO-SAT(*init*, *transition*, *goal*, t)
 $model \leftarrow$ SAT-SOLVER(cnf)
 if $model$ is not null **then**
 return EXTRACT-SOLUTION($model$)
 return *failure*



Planning Problem Translation

1. Propositionalize the actions

- For each action schema, form ground propositions by substituting constants for each of the variables

2. Add action exclusion axioms

- Assert that no two actions can occur at the same time

3. Add precondition axioms

- For each ground action A^t , add the axiom $A^t \Rightarrow PRE(A)^t$, i.e., if an action is taken at time t , then the preconditions must have been true

Planning Problem Translation

4. Define the initial state

- Assert F^0 for every fluent F in the problem initial state
- Assert $\neg F^0$ for every fluent not mentioned in the initial state

5. Propositionalize the goal

- Goal becomes a disjunction over all its ground instances; variables replaced by constants

6. Add successor-state axioms

- For each fluent F , add an axiom of the form

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNot F^t)$$

where $ActionCauesF$ stands for a disjunction of all the ground actions that add F and $ActionCausesNotF$ stands for a disjunction of all the ground actions that delete F

Example: Eat a Cake!

Init($\neg Have(Cake)$)

Goal($\neg Have(Cake) \wedge Eaten(Cake)$)

Action(Eat(Cake))

Precond: *Have(Cake)*

Effect: : $\neg Have(Cake) \wedge Eaten(Cake)$

Action(Bake(Cake))

Precond: $\neg Have(Cake)$

Effect: : *Have(Cake)*

Example: Eat a Cake!

Bounded planning problem with possible solution at $n = 2$

- Initial and goal states:

(Init) $\neg \text{Have}(\text{Cake}, 0)$

(Goal) $\neg \text{Have}(\text{Cake}, 2) \wedge \text{Eaten}(\text{Cake}, 2)$

- The actions:

(Eat) $\text{Eat}(\text{cake}, 0) \rightarrow \text{Have}(\text{Cake}, 0) \wedge \neg \text{Have}(\text{Cake}, 1) \wedge \text{Eaten}(\text{Cake}, 1)$

$\text{Eat}(\text{cake}, 1) \rightarrow \text{Have}(\text{Cake}, 1) \wedge \neg \text{Have}(\text{Cake}, 2) \wedge \text{Eaten}(\text{Cake}, 2)$

(Bake) $\text{Bake}(\text{cake}, 0) \rightarrow \neg \text{Have}(\text{Cake}, 0) \wedge \text{Have}(\text{Cake}, 1)$

$\text{Bake}(\text{cake}, 1) \rightarrow \neg \text{Have}(\text{Cake}, 1) \wedge \text{Have}(\text{Cake}, 2)$

Example: Eat a Cake! (cont.)

- **Successor state axioms** : (How fluents can change)

$\neg \text{Have}(\text{Cake}, 0) \wedge \text{Have}(\text{Cake}, 1)$	$\rightarrow \text{Bake}(\text{Cake}, 0)$
$\neg \text{Have}(\text{Cake}, 1) \wedge \text{Have}(\text{Cake}, 2)$	$\rightarrow \text{Bake}(\text{Cake}, 1)$
$\text{Have}(\text{Cake}, 0) \wedge \neg \text{Have}(\text{Cake}, 1)$	$\rightarrow \text{Eat}(\text{Cake}, 0)$
$\text{Have}(\text{Cake}, 1) \wedge \neg \text{Have}(\text{Cake}, 2)$	$\rightarrow \text{Eat}(\text{Cake}, 1)$
$\neg \text{Eaten}(\text{Cake}, 0) \wedge \text{Eaten}(\text{Cake}, 1)$	$\rightarrow \text{Eat}(\text{Cake}, 0)$
$\neg \text{Eaten}(\text{Cake}, 1) \wedge \text{Eaten}(\text{Cake}, 2)$	$\rightarrow \text{Eat}(\text{Cake}, 1) \dots$

- **Action exclusion axioms**: (Assume not eat and bake at same time!)

$\neg \text{Eat}(\text{Cake}, 0) \vee \neg \text{Bake}(\text{Cake}, 0)$
 $\neg \text{Eat}(\text{Cake}, 1) \vee \neg \text{Bake}(\text{Cake}, 1)$



The Frame Problem

- The Frame Problem:
 - Most actions leave most fluents unchanged.
 - For each action, assert an axiom for each fluent the action leaves unchanged.
- Successor-state axioms:
 - A clever encoding that handles the frame problem well.
 - For m actions and n fluents, need $O(mn)$ such axioms
 - In contrast, with successor-state axioms, only need $O(n)$ axioms: each axiom is longer, but only involves actions that have an effect on the fluent
- SATPlan
 - Find possible reachable models specifying future action sequences
 - Guaranteed to find shortest path if one exists
 - Works only for fully observable or sensorless environments
 - Can find models with impossible action; memory requirement is still combinatorially large



Summary

- Classical planning
 - Goal oriented action planning (STRIPS Planning)
 - State, action, goal representations in PDDL
- State-space search
 - Forward or progressive search
 - Backward or regressive search
- Planning as satisfiability
 - Planning as propositional satisfiability problem
 - The SATPlan algorithm
- Other classical planning approaches
 - **Planning graph** – specialized data structure to encode constraints and derive heuristics
 - **Situation calculus** – describing planning problem in first-order logic
 - **Constraint satisfaction problems (CSP)** – encode bounded planning problem
 - **Partial-order planning** – represents plan as a graph; handles independent subproblems, explainability



Complexity of Planning



Complexity of Planning

- Two decision problems:
 - **PlanSAT** –whether there exists any plan that solves a planning problem
 - **Bounded PlanSAT** –whether there is a solution of length k or less
 - For classical planning, both problems are decidable as number of states is finite
- In general:
 - Both problems are in PSPACE: solvable with polynomial amount of space
 - In many domains, Bounded PlanSAT is NP-complete; PlanSAT is in P
 - Optimal planning is usually hard, but sub-optimal planning is sometimes easy
- In real world planning:
 - Agents usually asked to find plans in specific domains, not in worst-case instances
 - To do well, need good **heuristics** or **alternate planning models**



Homework

- Readings:

- Classical planning: RN: 11.1, 11.2
- SATPlan: RN 7.7.4, 11.2.3
- Complexity: RN Chapter 11, Bibliographical and Historical Notes (last page)

- Reviews:

- Search (review): RN 3.3, 3.4, 3.5
- Proposition logic and inference (review): RN: 7.3, 7.4, 7.5