Classical Planning

CS4246/CS5446

AI Planning and Decision Making

This lecture will be recorded!

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
- Planning Algorithm
  - Input: a problem
  - Output: a solution in the form of an action sequence (a plan)
- Planning Solution
  - A plan or path from the initial state(s) to the goal state(s)
    - Any path
  - A goal state that satisfies certain properties

Classical Planning

Real-world example:
Scheduling classes in NUS

- 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?

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

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	$At(p, SIN) \wedge Plane(p)$

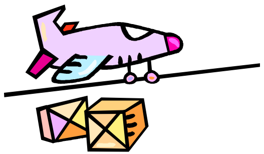
Main Characteristics

- 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 $Plane_2$

Exercise

- Which of the following are valid fluents allowed in a state?

  A: $At(x, SR2)$

  B: $At(BestFriend(WinniePhePooh), HundredAcreWoods)$

Quiz

Quiz answer

Quiz answer

Goal Representation

- Goal State:
  - Goal g is a partially specified state, represented as a conjunction of literals
  - A state s satisfies a goal g if s contains all the literals in g
  - Examples:
    - $Hungry \land Sleepy \land Bored$ satisfies the goal $Hungry \land Bored$
    - $At(Cargo_1, SFO)$ satisfies the goal $At(c, SFO)$ with substitution $\{c/Cargo_1\}$ where c is a variable for any cargo

Follow general definition of Goal in PDDL on Slide 21

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

$$
\begin{aligned}
&Action(Fly(p, from, to) \\
&\quad \text{PRECOND: } At(p, from) \land Plane(p) \land Airport(from) \land Airport(to) \\
&\quad \text{EFFECT: } \neg At(p, from) \land At(p, to))
\end{aligned}
$$

Grounded Actions

$Action(Fly(p, from, to)$
Precond: $At(p, from) \land Plane(p) \land Airport(from) \land Airport\ (to)$
Effect: $\neg At(p, from) \land At(p, to))$

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

$Action(Fly(P_1, SFO, SIN)$
PRECOND: $At(P_1, SFO) \land Plane(P_1) \land Airport(SFO) \land Airport\ (SIN)$
EFFECT: $\neg At(P_1, SFO) \land At(P_1, SIN))$

$Action(Fly(P_2, SIN, SFO)$
PRECOND: $At(P_2, SIN) \land Plane(P_2) \land Airport(SIN) \land Airport\ (SFO)$
EFFECT: $\neg At(P_2, SIN) \land At(P_2, SFO))$

Action Execution

- Current state and available action:

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

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

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

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

- Execute action

  $Fly(P_1, SFO, SIN)$

- Remove from database of states

  $At(P_1, SFO)$

- Add to database of states

  $At(P_1, SIN)$

- Updated state:

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

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

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?

  $At(P_1;\ SFO) \land Plane(P_1) \land Airport(SFO) \land Airport(SIN)$

  $Action(Fly(p;\ from;\ to)$

  PRECOND: $At(p;\ from)\ \land\ Plane(p) \land Airport(from) \land Airport(to)$

  EFFECT: $\neg At(p;\ from) \land At(p;\ to))$

  A. $At(P_1, SFO) \land Plane(P_1) \land Airport(SFO) \land Airport(SIN)$

  B. $At(P_1, SIN) \land Plane(P_1) \land Airport(SFO) \land Airport(SIN)$

  C. Not applicable.

The Frame Problem

	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	$At(p, SIN) \wedge Plane(p)$

- 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: $A \wedge t(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(SIN) \wedge \cdots$
- 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

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

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 $Fly(P_1, SIN, SIN)$?

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

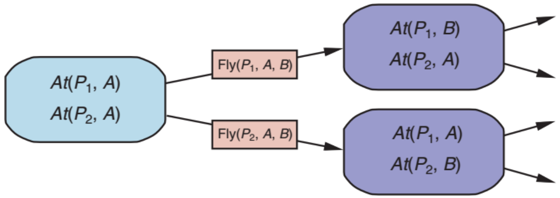
Forward (progression) search

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
  - Different planning procedures have different search spaces
- 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

Planning by Forward Search

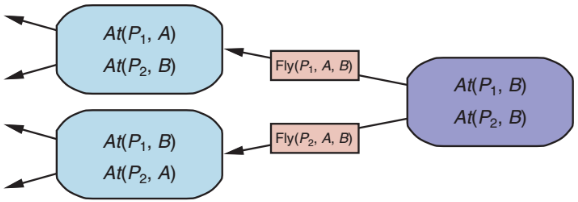


Start at a state s

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

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

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:

$$Pos(g') = (Pos(g) - Add(a) \cup Pos(Precond(a))$$
$$Neg(g') = (Neg(g) - Del(a) \cup Neg(Precond(a))$$

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: $In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
  - Effect: $At(c, a) \land \neg In(c, p)$
- New action schema after substitution
  - $Action(Unload(C_2, p', SFO)$
  - Precond: $In(C_2, p') \land At(p', SFO) \land Cargo(C_2) \land Plane(p') \land. Airport(SFO)$
  - Effect: $At(C_2, SFO) \land \neg In(C_2, p')$
- Regressed state (new goal)
  - $g' = In(C_2, p') \land At(p', SFO) \land Cargo(C_2) \land Plane(p') \land Airport(SFO)$

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

POS(g') = (POS(g) – ADD(a)) U POS(Precond(a))
NEG(g') = (NEG(g) – DEL(a)) U NEG(Precond(a))

Exercise

- 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?

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:

$$(P \lor Q) \land (\neg Q \lor R \lor S) \land (\neg R \lor \neg P)$$

    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

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*

Assignment of values to variables

Source: RN Fig 7.22

Example: Eat a Cake!

$Init\big(\neg Have(Cake)\big)$

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

$Action(Eat(Cake)$
Precond: $Have(Cake)$
Effect: $\neg Have(Cake) \land Eaten(Cake))$

$Action(Bake(Cake)$
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) $\neg Have(Cake, 0)$

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

- The actions:

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

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

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

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

Example: Eat a Cake! (cont.)

- Successor state axioms : (How fluents can change)

$\neg Have(Cake, 0) \wedge Have(Cake, 1) \rightarrow Bake(Cake, 0)$

$\neg Have(Cake, 1) \wedge Have(Cake, 2) \rightarrow Bake(Cake, 1)$

$Have (Cake, 0) \wedge \neg Have(Cake, 1) \rightarrow Eat(Cake, 0)$

$Have (Cake, 1) \wedge \neg Have(Cake, 2) \rightarrow Eat(Cake, 1)$

$\neg Eaten (Cake, 0) \wedge Eaten(Cake, 1) \rightarrow Eat(Cake, 0)$

$\neg Eaten (Cake, 1) \wedge Eaten(Cake, 2) \rightarrow Eat(Cake, 1)$ …

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

$\neg Eat (Cake, 0) \vee \neg Bake(Cake, 0)$

$\neg Eat (Cake, 1) \vee \neg Bake(Cake, 1)$

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 are replaced by constants
6. Add successor state axioms
   - For each fluent F, add an axiom of the form

$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t)$, where $ActionCausesF^t$ stands for a disjunction of all the ground actions that add F and $ActionCausesNotF^t$ stands for a disjunction of all the ground actions that delete F

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

Characteristics of Planning

Planning Algorithm Properties

- A planning algorithm or planner is sound
  - For any plan generated by the planning algorithm, this plan is guaranteed to be a solution
- A planning algorithm or planner is complete
  - If a solution exists then the planning algorithm will return a solution
- Question: How "good" are the solutions?
  - For any plan generated by the planning algorithm, can the plan be guaranteed to be an optimal solution?
  - Yes if admissible heuristics are used in the search for the solution

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

Classical Planning
(Additional slides)

CS4246/CS5446

AI Planning and Decision Making

Required Background

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

  $Unify\big(Brother(John, x), Brother(y, James)\big)$
  $= \{x/James, y/John\}$
  - For every pair of unifiable sentences, there is a most general unifier (MGU), which is unique up to renaming and substitution of variables

Review: Search

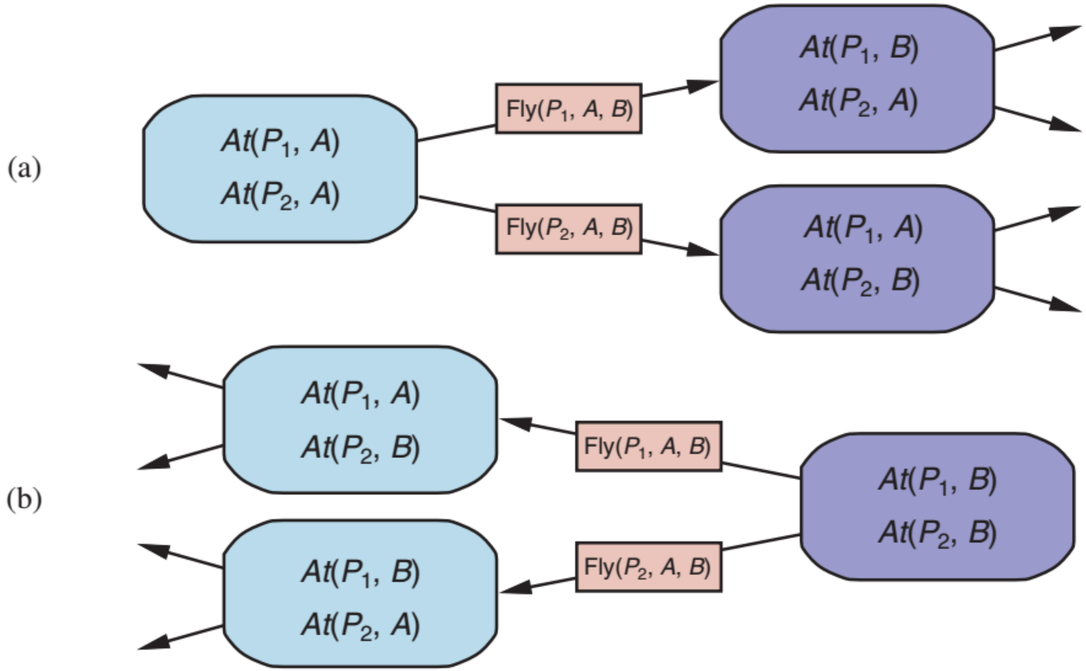
- Uniform cost search
  - Breadth-first search
  - Depth-first search
  - Iterative-deepening search
- Heuristic search
  - Greedy best-first search
  - A* search
  - IDA* search
- Deriving heuristics
  - Admissible heuristics
  - Consistent heuristics
  - Relaxed problems

Planning as State-Space Search Examples

Forward (progression) search examples

Backward (regression) search examples

Forward and Backward Search



Source: RN Figure 11.5

Example: Honey on Table

- Initial state
  - On(EmptyPot, Table) ∧ On(FullPot, Shelf)
  - Pot(FullPot) ∧ Pot(EmptyPot) ∧ Place(Shelf) ∧ Place(Table)
- Goal
  - On(FullPot, Table)
- Actions
  - **Get**(FullPot, place)

    Precond: On(FullPot, place)

    Effect: ¬On(FullPot, place) ∧ In(FullPot, Hand)
  - **Get**(EmptyPot, place)

    Precond: On(EmptyPot, place)

    Effect: ¬On(EmptyPot, place) - This goes straight to the Trash, don't hold in Hand
  - **Put**(pot, place)

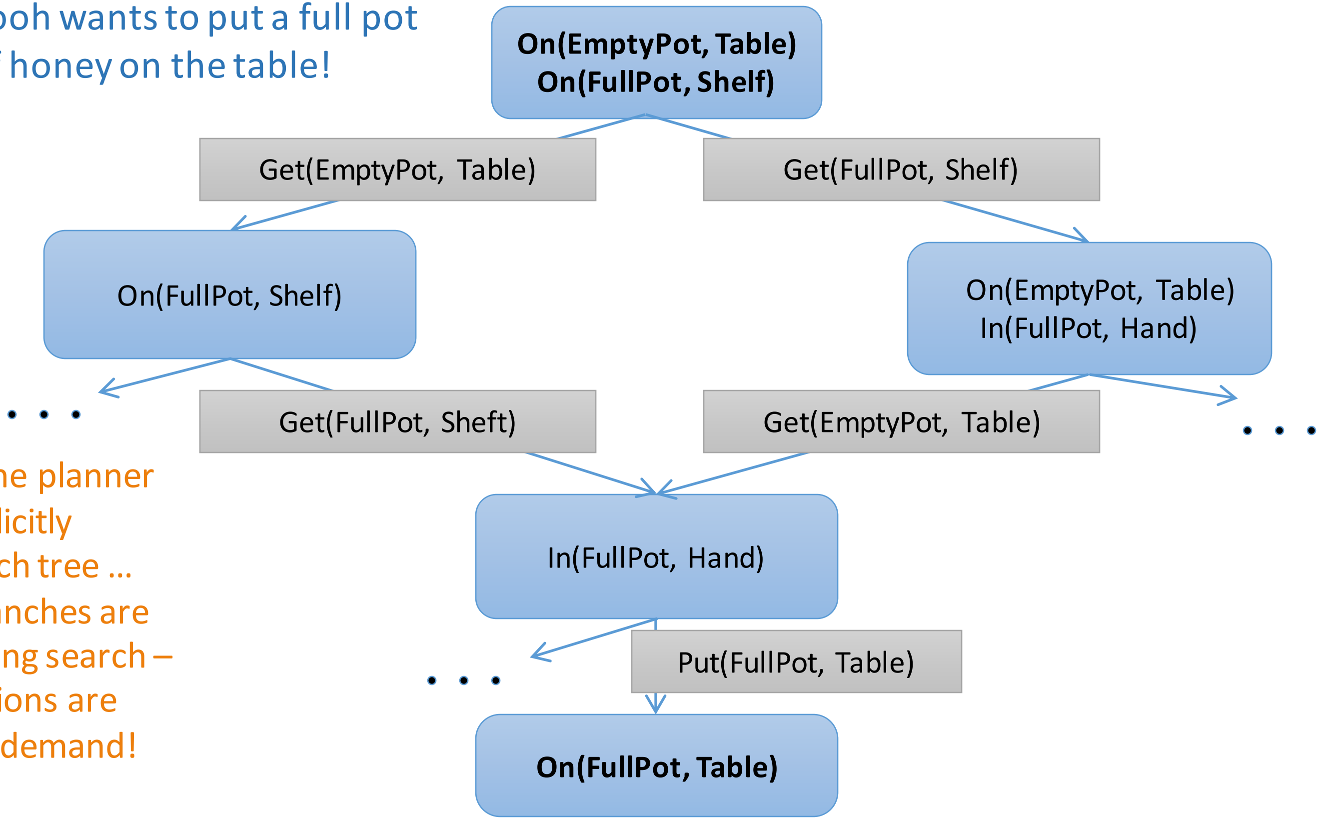
    Precond: ¬On(pot, place) ∧ In(pot, Hand)

    Effect: On(pot, place) ∧ ¬In(pot, Hand)

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

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!

Example: Forward Search

Goal: On(FullPot, Table)

Start at initial state

Is goal satisfied?

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

Example: Forward Search

Goal: On(FullPot, Table)

Compute Applicable Actions

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

Get(EmptyPot, place)
Precond: On(EmptyPot, place)
Effect: $\neg$On(EmptyPot, place)
Subst = {place/Table}

Get(EmptyPot, Table)

Get(FullPot, Shelf)

Get(FullPot, place)
Precond: On(FullPot, place)
Effect: $\neg$On(FullPot, place) $\wedge$ In(FullPot, Hand)
Subst = {place/Shelf}

Example: Forward Search

Goal: On(FullPot, Table)

Compute Successor States

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

Get(EmptyPot, place)
Precond: On(EmptyPot, place)
Effect: ¬On(EmptyPot, place)
Subst = {place/Table}

Get(EmptyPot, Table)

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

Get(FullPot, place)
Precond: On(FullPot, place)
Effect: ¬On(FullPot, place) ∧ In(FullPot, Hand)
Subst = {place/Shelf}

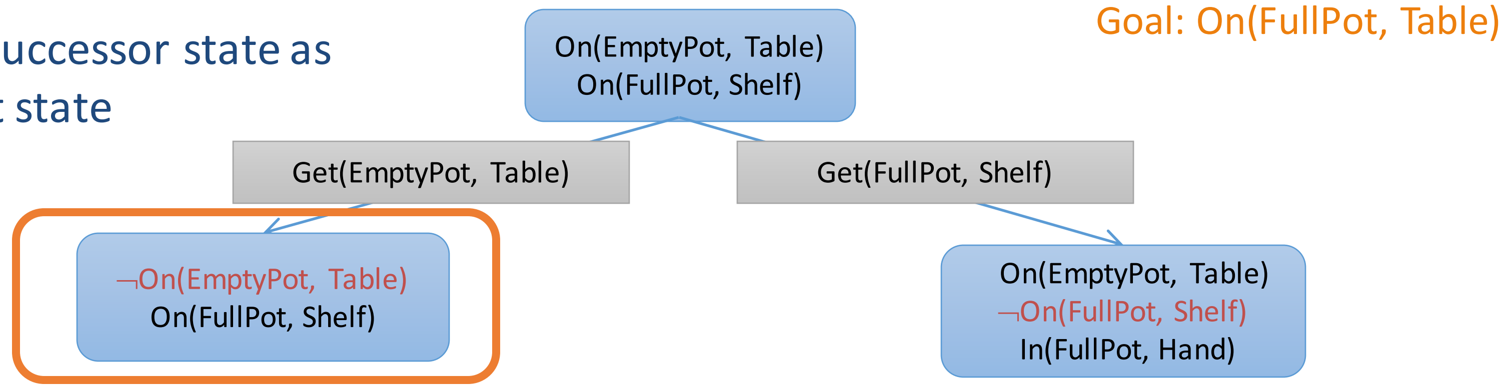
Get(FullPot, Shelf)

On(EmptyPot, Table)
¬On(FullPot, Shelf)
In(FullPot, Hand)

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

Example: Forward Search

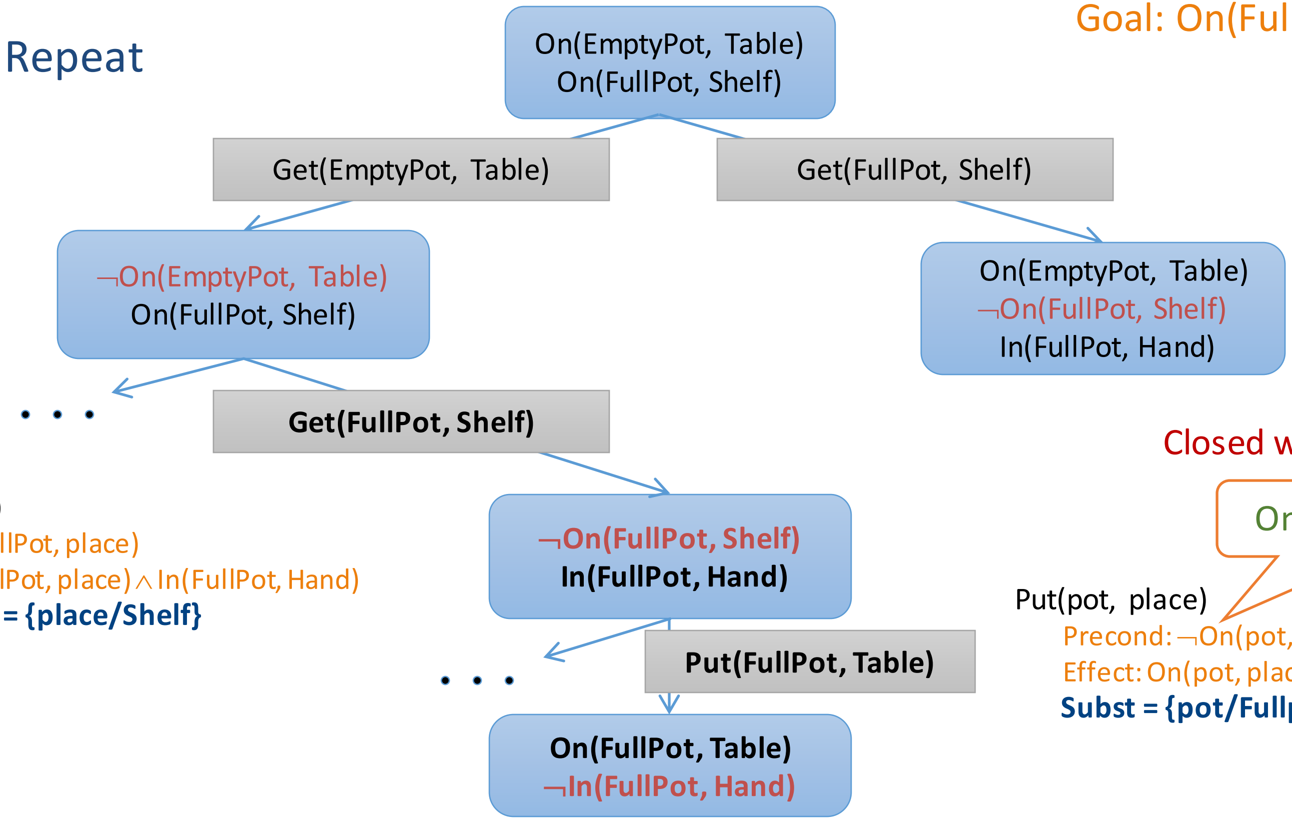
Pick a successor state as current state



Example: Forward Search

Goal: On(FullPot, Table)

Repeat



Get(FullPot, place)
 Precond: On(FullPot, place)
 Effect: ¬On(FullPot, place) ∧ In(FullPot, Hand)
 Subst = {place/Shelf}

Closed world assumption!

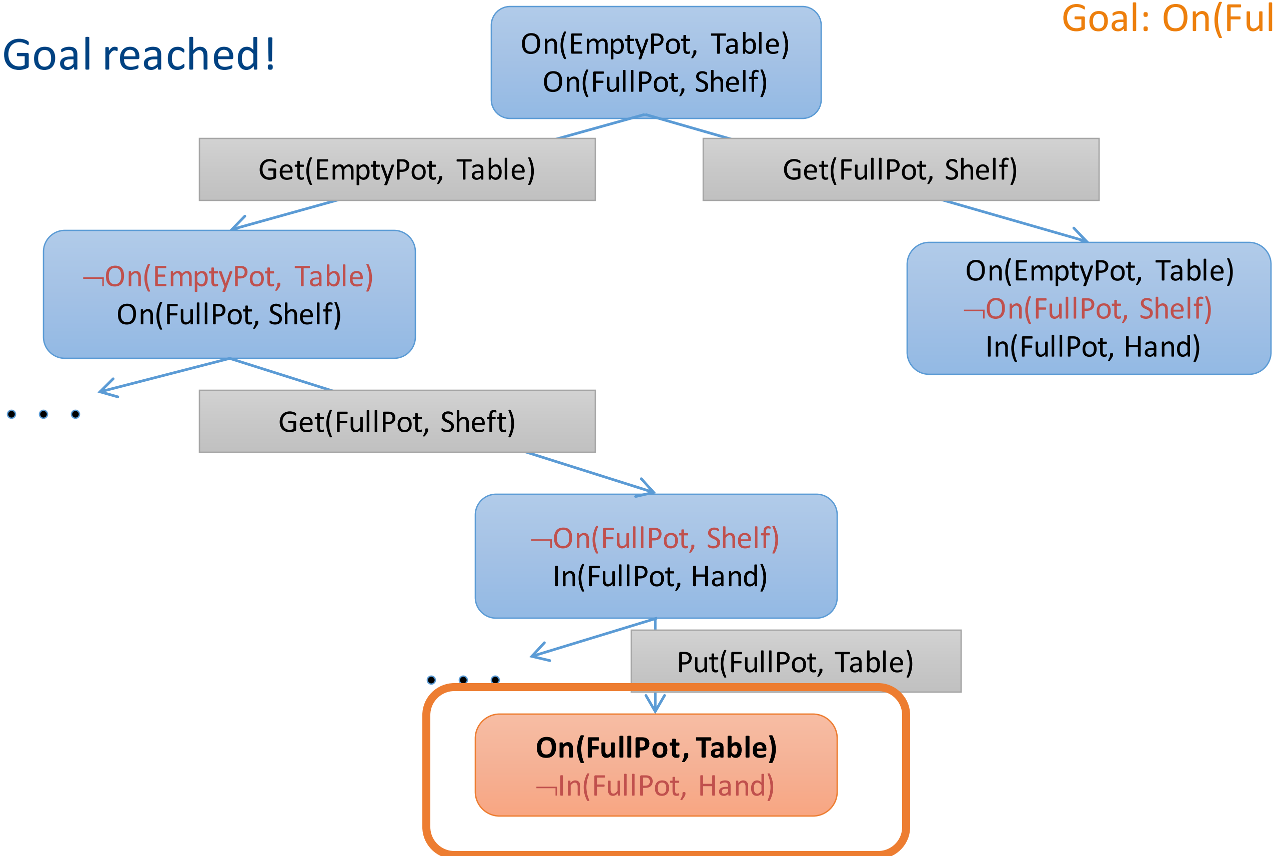
On(FullPot, Table)?

Put(pot, place)
 Precond: ¬On(pot, place) ∧ In(pot, Hand)
 Effect: On(pot, place) ∧ ¬In(pot, Hand)
 Subst = {pot/Fullpot, place/Table}

Example: Forward Search

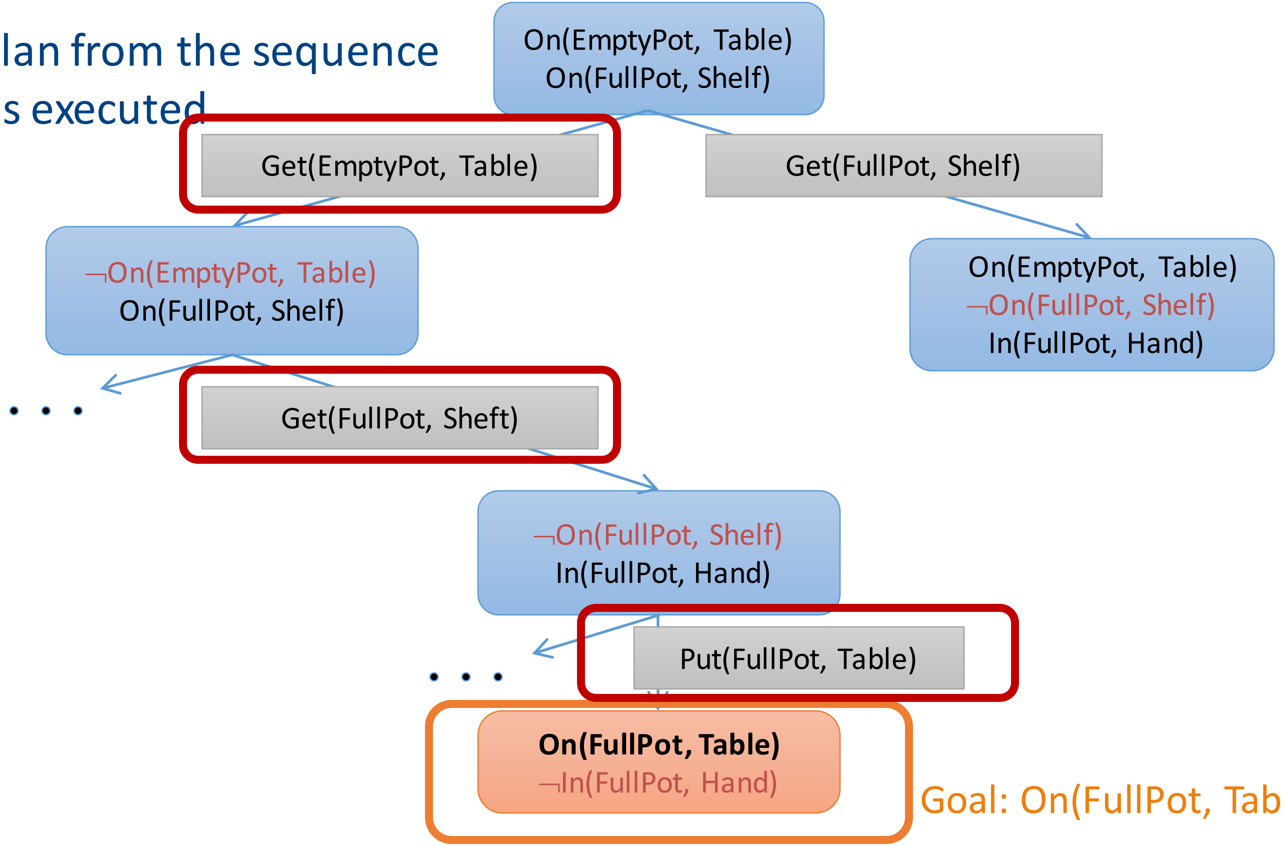
Goal: On(FullPot, Table)

Success! Goal reached!



Example: Forward Search

Extract plan from the sequence of actions executed



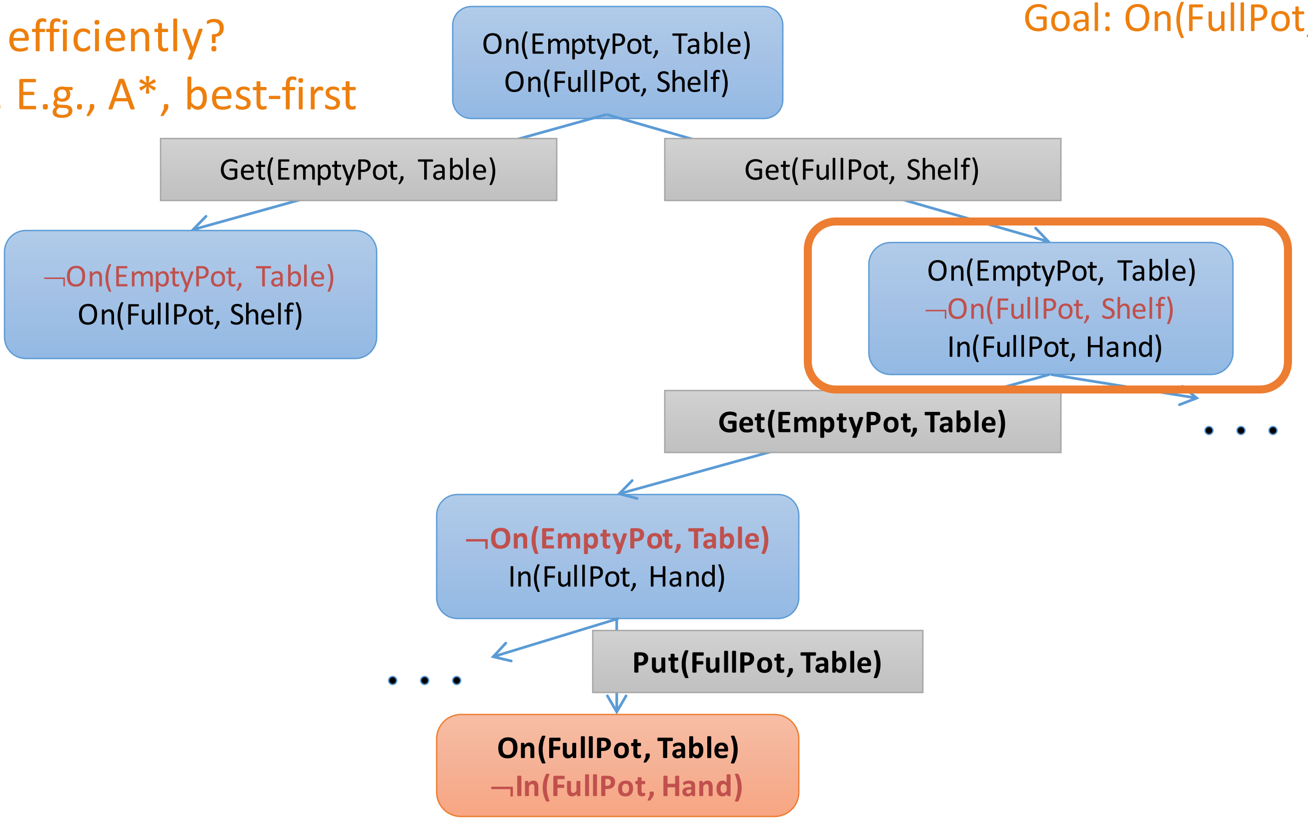
Goal: On(FullPot, Table)

Example: Forward Search – Alternate Path

How to search efficiently?

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

Goal: On(FullPot, Table)



Example: Backward Search

Start at goal state

Is initial state satisfied?

Initial state:

On(EmptyPot, Table) $\wedge$ On(FullPot, Shelf)

On(FullPot, Table)

Example: Backward Search

Compute relevant actions

Initial state:
On(EmptyPot, Table) $\wedge$ On(FullPot, Shelf)

Put(FullPot, Table)

On(FullPot, Table)

Put(pot, place)
Precond: $\neg$On(pot, place) $\wedge$ In(pot, Hand)
Effect: On(pot, place) $\wedge$ $\neg$In(pot, Hand)
Subst = {pot/Fullpot, place/Table}

Example: Backward Search

Compute predecessor states

Initial state:
On(EmptyPot, Table) $\wedge$ On(FullPot, Shelf)

POS(g') = (POS(g) – ADD(a)) U POS(Precond(a))
NEG(g') = (NEG(g) – DEL(a)) U NEG(Precond(a))

$\neg$On(FullPot, Table)
In(FullPot, Hand)

Put(FullPot, Table)

On(FullPot, Table)

Put(Fullpot, Table)
Precond: $\neg$On(Fullpot, Table) $\wedge$ In(Fullpot, Hand)
Effect: On(Fullpot, Table) $\wedge$ $\neg$In(Fullpot, Hand)

Put(pot, place)
Precond: $\neg$On(pot, place) $\wedge$ In(pot, Hand)
Effect: On(pot, place) $\wedge$ $\neg$In(pot, Hand)
Subst = {pot/Fullpot, place/Table}

Example: Backward Search

Pick a predecessor state
as the current goal state

Initial state:
On(EmptyPot, Table) $\wedge$ On(FullPot, Shelf)

Get(FullPot, Shelf)

Get(EmptyPot, Table)

Get(FullPot, place)
Precond: On(FullPot, place)
Effect: $\neg$On(FullPot, place) $\wedge$ In(FullPot, Hand)
Subst = {place/Shelf}

$\neg$On(FullPot, Table)
In(FullPot, Hand)

Get(EmptyPot, place)
Precond: On(EmptyPot, place)
Effect: $\neg$On(EmptyPot, place)
Subst = {place/Table}

Put(FullPot, Table)

On(FullPot, Table)

Example: Backward Search

Repeat

. . .

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

Initial state:
On(EmptyPot, Table) $\wedge$ On(FullPot, Shelf)

Get(EmptyPot, Table)

Get(EmptyPot, place)
Precond: On(EmptyPot, place)
Effect: $\neg$On(EmptyPot, place)
Subst = {place/Table}

On(FullPot, Shelf)

Get(FullPot, Shelf)

Get(EmptyPot, Table)

Get(FullPot, place)
Precond: On(FullPot, place)
Effect: $\neg$On(FullPot, place) $\wedge$ In(FullPot, Hand)
Subst = {place/Shelf}

$\neg$On(FullPot, Shelf)
$\neg$On(FullPot, Table)
In(FullPot, Hand)

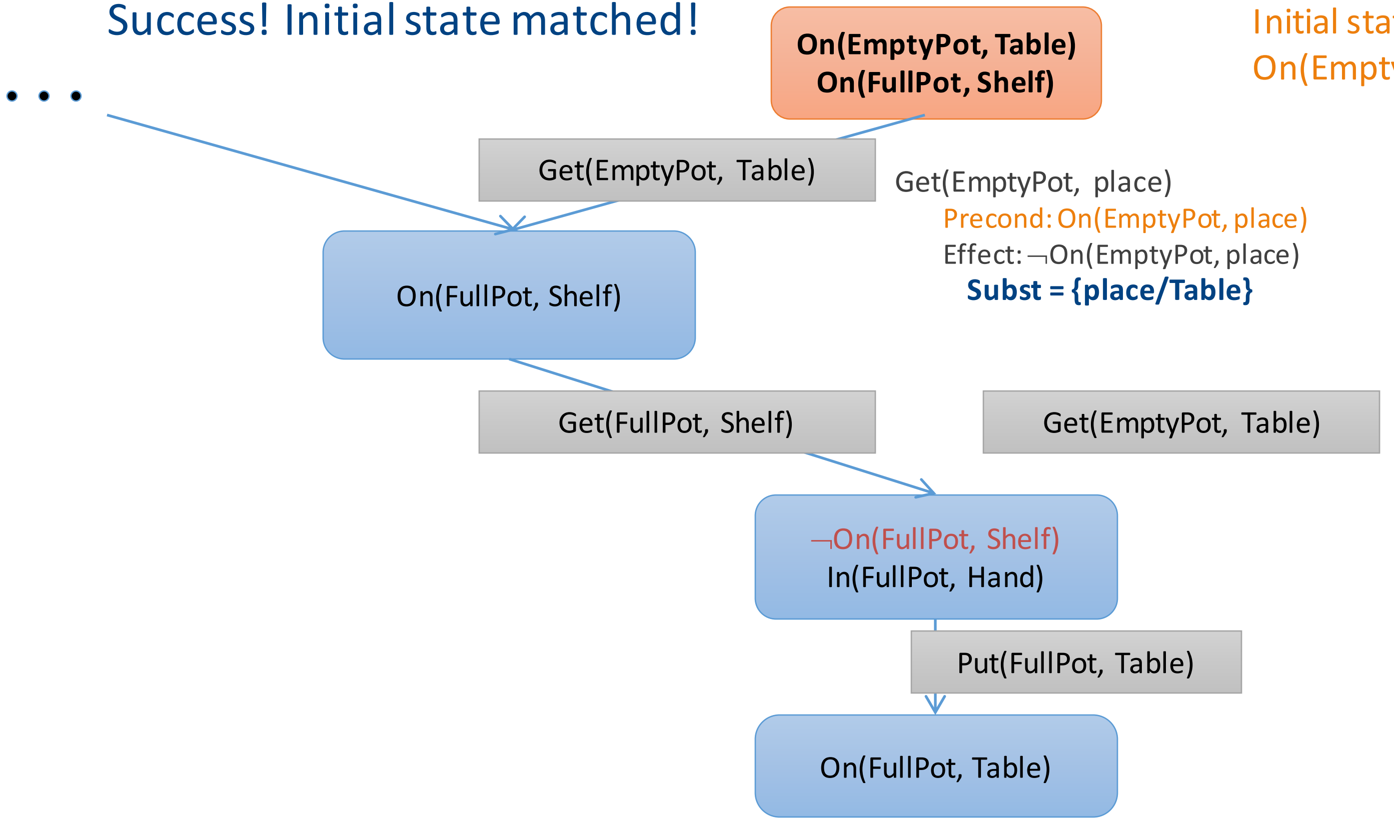
Put(FullPot, Table)

On(FullPot, Table)

Example: Backward Search

Success! Initial state matched!

Initial state:
On(EmptyPot, Table) $\wedge$ On(FullPot, Shelf)



Get(EmptyPot, place)
Precond: On(EmptyPot, place)
Effect: ¬On(EmptyPot, place)
Subst = {place/Table}

Example: Backward Search – Alternate Path

Success! Initial state matched!